目錄

spark机器学习算法研究和源码分析

本项目对 `spark ml` 包中各种算法的原理加以介绍并且对算法的代码实现进行详细分析，旨在加深自己对机器学习算法的理解，熟悉这些算法的分布式实现方式。

本系列文章支持的spark版本

- **spark 2.x**

本系列的目录结构

本系列目录如下：

- 数据类型
- 基本统计
  - summary statistics（概括统计）
  - correlations（相关性系数）
  - tratified sampling（分层取样）
  - hypothesis testing（假设检验）
  - random data generation（随机数生成）
  - Kernel density estimation（核密度估计）
- 协同过滤
  - 交换最小二乘
- 分类和回归
  - 线性模型
    - SVMs(支持向量机)
    - 逻辑回归
    - 线性回归
  - 朴素贝叶斯
  - 决策树
  - 组合树
    - 随机森林
    - 梯度提升树

        - MaxAbsScaler
        - Bucketizer
        - ElementwiseProduct(元素智能乘积)
        - SQLTransformer
        - VectorAssembler
        - QuantileDiscretizer
    - 特征选择
        - VectorSlicer
        - RFormula
        - ChiSqSelector(卡方选择器)

说明

本专题的大部分内容来自spark源码、spark官方文档，并不用于商业用途。转载请注明本专题地址。 本专题引用他人的内容均列出了参考文献，如有侵权，请务必邮件通知作者。邮箱地址： `endymecy@sina.cn` 。

本专题的部分文章中用到了latex来写数学公式,可以在浏览器中安装 `MathJax` 插件用来展示这些公式。

本人水平有限，分析中难免有错误和误解的地方，请大家不吝指教，万分感激。有问题可以到 Q&A ask question 讨论。

License

本文使用的许可见 LICENSE

数据类型

`MLlib` 既支持保存在单台机器上的本地向量和矩阵，也支持备份在一个或多个 `RDD` 中的分布式矩阵。本地向量和本地矩阵是简单的数据模型，作为公共接口提供。底层的线性代数操作通过Breeze和jblas提供。 在 `MLlib` 中，用于有监督学习的训练样本称为标注点(`labeled point`)。

1 本地向量(Local vector)

一个本地向量拥有从0开始的 `integer` 类型的索引以及 `double` 类型的值，它保存在单台机器上面。 `MLlib` 支持两种类型的本地向量：稠密(`dense`)向量和稀疏(`sparse`)向量。 一个稠密向量通过一个 `double` 类型的数组保存数据，这个数组表示向量的条目值(`entry values`)；一个稀疏向量通过两个并行的数组（ `indices和values` ）保存数据。例如，一个向量 `(1.0, 0.0, 3.0)` 可以以稠密的格式保存为 `[1.0, 0.0, 3.0]` 或者以稀疏的格式保存为 `(3, [0, 2], [1.0, 3.0])` ，其中3表示数组的大小。

本地向量的基类是Vector， `Spark` 提供了两种实现： DenseVector和SparseVector。 `Spark` 官方推荐使用Vectors中实现的工厂方法去创建本地向量。下面是创建本地向量的例子。

```
import org.apache.spark.mllib.linalg.{Vector, Vectors}
// 创建一个dense vector (1.0, 0.0, 3.0).
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
// 创建一个sparse vector (1.0, 0.0, 3.0)并且指定它的索引和值
val sv1: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0)
)
// 创建一个sparse vector (1.0, 0.0, 3.0)并且指定它的索引和值
val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
```

注意， `Scala` 默认引入 `scala.collection.immutable.Vector` ，这里我们需要主动引入 `MLLib` 中的 `org.apache.spark.mllib.linalg.Vector` 来操作。我们可以看看 `Vectors` 对象的部分方法。

```
def dense(firstValue: Double, otherValues: Double*): Vector =
    new DenseVector((firstValue +: otherValues).toArray)
def dense(values: Array[Double]): Vector = new DenseVector(value
s)
def sparse(size: Int, indices: Array[Int], values: Array[Double]
): Vector =
    new SparseVector(size, indices, values)
def sparse(size: Int, elements: Seq[(Int, Double)]): Vector = {
    require(size > 0, "The size of the requested sparse vector m
ust be greater than 0.")
    val (indices, values) = elements.sortBy(_._1).unzip
    var prev = -1
    indices.foreach { i =>
      require(prev < i, s"Found duplicate indices: $i.")
      prev = i
    }
    require(prev < size, s"You may not write an element to index
$prev because the declared " +
      s"size of your vector is $size")
    new SparseVector(size, indices.toArray, values.toArray)
  }
```

2 标注点(Labeled point)

一个标注点就是一个本地向量（或者是稠密的或者是稀疏的），这个向量和一个标签或者响应相关联。在 `MLlib` 中，标注点用于有监督学习算法。我们用一个 `double` 存储标签，这样我们就可以在回归和分类中使用标注点。 对于二分类，一个标签可能是0或者是1；对于多分类，一个标签可能代表从0开始的类别索引。

在 `MLlib` 中，一个标注点通过样本类LabeledPoint表示。

```
@Since("0.8.0")
@BeanInfo
case class LabeledPoint @Since("1.0.0") (
    @Since("0.8.0") label: Double,
    @Since("1.0.0") features: Vector) {
  override def toString: String = {
    s"($label,$features)"
  }
}
```

下面是使用 `LabeledPoint` 的一个例子。

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
// Create a labeled point with a positive label and a dense feat
ure vector.
val pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))
// Create a labeled point with a negative label and a sparse fea
ture vector.
val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0, 2), Array(
1.0, 3.0)))
```

在现实的应用中，训练数据是稀疏的情况非常常见， `MLlib` 支持读取训练数据存储为 `LIBSVM` 格式。它是LIBSVM和LIBLINEAR默认的格式。 它是一种文本格式，每一行表示一个标注的稀疏特征向量，如下所示：

```
label index1:value1 index2:value2 ...
```

3 本地矩阵（Local matrix）

一个本地矩阵拥有 `Integer` 类型的行和列索引以及 `Double` 类型的值。 `MLlib` 支持稠密矩阵和稀疏矩阵两种。稠密矩阵将条目(`entry`)值保存为单个 `double` 数组，这个数组根据列的顺序存储。 稀疏矩阵的非零条目值保存为压缩稀疏列（ `Compressed Sparse Column ，CSC` ）格式，这种格式也是以列顺序存储。例如下面的稠密矩阵：

$$\begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ 5.0 & 6.0 \end{pmatrix}$$

这个稠密矩阵保存为一维数组 `[1.0, 3.0, 5.0, 2.0, 4.0, 6.0]` ，数组大小为 `(3,2)` 。

本地矩阵的基类是Matrix，它提供了两种实现：DenseMatrix和SparseMatrix。推荐使用Matrices的工厂方法来创建本地矩阵。下面是一个实现的例子：

```
import org.apache.spark.mllib.linalg.{Matrix, Matrices}
// Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0,
4.0, 6.0))
// Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))
val sm: Matrix = Matrices.sparse(3, 2, Array(0, 1, 3), Array(0, 2
, 1), Array(9, 6, 8))
```

稠密矩阵的存储很简单，不赘述。稀疏矩阵的存储使用 `CSC` 。关于压缩矩阵的介绍，请参看文献【1】。

4 分布式矩阵(Distributed matrix)

一个分布式矩阵拥有 `long` 类型的行和列索引，以及 `double` 类型的值，分布式的存储在一个或多个 `RDD` 中。选择正确的格式存储大型分布式矩阵是非常重要的。将一个分布式矩阵转换为另外一个格式可能需要一个全局的 `shuffle` ，这是非常昂贵的。 到目前为止，已经实现了三种类型的分布式矩阵。

基本的类型是 `RowMatrix` ， `RowMatrix` 是一个面向行的分布式矩阵，它没有有意义的行索引。它的行保存为一个 `RDD` ,每一行都是一个本地向量。我们假设一个 `RowMatrix` 的列的数量不是很巨大，这样单个本地向量可以方便的和 `driver` 通信，也可以被单个节点保存和操作。 `IndexedRowMatrix` 和 `RowMatrix` 很像，但是它拥有行索引，行索引可以用于识别行和进行 `join` 操作。 `CoordinateMatrix` 是一个分布式矩阵，它使用 `COO` 格式存储。请参看文献【1】了解 `COO` 格式。

4.1 RowMatrix

`RowMatrix` 是一个面向行的分布式矩阵，它没有有意义的行索引。它的行保存为一个 `RDD` ,每一行都是一个本地向量。因为每一行保存为一个本地向量，所以列数限制在了整数范围。

一个 `RowMatrix` 可以通过 `RDD[Vector]` 实例创建。创建完之后，我们可以计算它的列的统计和分解。QR分解的形式为 `A=QR` ，其中 `Q` 是一个正交矩阵， `R` 是一个上三角矩阵。下面是一个 `RowMatrix` 的例子。

```
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.mllib.linalg.distributed.RowMatrix
val rows: RDD[Vector] = ... // an RDD of local vectors
// Create a RowMatrix from an RDD[Vector].
val mat: RowMatrix = new RowMatrix(rows)
// Get its size.
val m = mat.numRows()
val n = mat.numCols()
// QR decomposition
val qrResult = mat.tallSkinnyQR(true)
```

4.2 IndexedRowMatrix

`IndexedRowMatrix` 和 `RowMatrix` 很像，但是它拥有行索引。索引的行保存为一个 `RDD[IndexedRow]` ，其中 `IndexedRow` 是一个参数为 `(Long, Vector)` 的样本类，所以每一行通过它的索引以及一个本地向量表示。

一个 `IndexedRowMatrix` 可以通过 `RDD[IndexedRow]` 实例创建，并且一个 `IndexedRowMatrix` 可以通过去掉它的行索引，转换成 `RowMatrix` 。下面是一个例子：

```
import org.apache.spark.mllib.linalg.distributed.{IndexedRow, In
dexedRowMatrix, RowMatrix}
val rows: RDD[IndexedRow] = ... // an RDD of indexed rows
// Create an IndexedRowMatrix from an RDD[IndexedRow].
val mat: IndexedRowMatrix = new IndexedRowMatrix(rows)
// Get its size.
val m = mat.numRows()
val n = mat.numCols()
// Drop its row indices.
val rowMat: RowMatrix = mat.toRowMatrix()
```

`IndexedRow` 这个样本类的代码如下：

```
case class IndexedRow(index: Long, vector: Vector)
```

4.3 CoordinateMatrix

`CoordinateMatrix` 是一个分布式矩阵，它的条目保存为一个 `RDD` 。每一个条目是一个 `(i: Long, j: Long, value: Double)` 格式的元组，其中 `i` 表示行索引， `j` 表示列索引， `value` 表示条目值。 `CoordinateMatrix` 应该仅仅在矩阵维度很大并且矩阵非常稀疏的情况下使用。

`CoordinateMatrix` 可以通过 `RDD[MatrixEntry]` 实例创建，其中 `MatrixEntry` 是 `(Long, Long, Double)` 的包装。 `CoordinateMatrix` 可以转换成 `IndexedRowMatrix` 。下面是一个例子：

```
import org.apache.spark.mllib.linalg.distributed.{CoordinateMatr
ix, MatrixEntry}
val entries: RDD[MatrixEntry] = ... // an RDD of matrix entries
// Create a CoordinateMatrix from an RDD[MatrixEntry].
val mat: CoordinateMatrix = new CoordinateMatrix(entries)
// Get its size.
val m = mat.numRows()
val n = mat.numCols()
// Convert it to an IndexRowMatrix whose rows are sparse vectors.

val indexedRowMatrix = mat.toIndexedRowMatrix()
```

`MatrixEntry` 这个样本类的代码如下：

```
case class MatrixEntry(i: Long, j: Long, value: Double)
```

4.4 BlockMatrix

`BlockMatrix` 是一个分布式矩阵，它的保存为一个 `MatrixBlocks` 的 `RDD` 。 `MatrixBlock` 是一个 `((Int, Int), Matrix)` 类型的元组，其中 `(Int, Int)` 代表块的索引， `Matrix` 代表子矩阵。 `BlockMatrix` 支持诸如 `add` 和 `multiply` 等方法。 `BlockMatrix` 还有一个帮助方法 `validate` ，用来判断一个 `BlockMatrix` 是否正确的创建。

可以轻松的通过调用 `toBlockMatrix` 从一个 `IndexedRowMatrix` 或者 `CoordinateMatrix` 创建一个 `BlockMatrix` 。 `toBlockMatrix` 默认创建 `1024 * 1024` 大小的块，用户可以手动修个块的大小。 下面是一个例子：

```
import org.apache.spark.mllib.linalg.distributed.{BlockMatrix, C
oordinateMatrix, MatrixEntry}
val entries: RDD[MatrixEntry] = ... // an RDD of (i, j, v) matri
x entries
// Create a CoordinateMatrix from an RDD[MatrixEntry].
val coordMat: CoordinateMatrix = new CoordinateMatrix(entries)
// Transform the CoordinateMatrix to a BlockMatrix
val matA: BlockMatrix = coordMat.toBlockMatrix().cache()
// Validate whether the BlockMatrix is set up properly. Throws a
n Exception when it is not valid.
// Nothing happens if it is valid.
matA.validate()
// Calculate A^T A.
val ata = matA.transpose.multiply(matA)
```

5 参考文献

【1】稀疏矩阵存储格式总结+存储效率对比:COO,CSR,DIA,ELL,HYB

概括统计

`MLlib`支持`RDD[Vector]`列的概括统计，它通过调用`Statistics`的`colStats`方法实现。`colStats`返回一个`MultivariateStatisticalSummary`对象，这个对象包含列式的最大值、最小值、均值、方差等等。下面是一个应用例子：

```
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.mllib.stat.{MultivariateStatisticalSumma
ry, Statistics}
val observations: RDD[Vector] = ... // an RDD of Vectors
// Compute column summary statistics.
val summary: MultivariateStatisticalSummary = Statistics.colStat
s(observations)
println(summary.mean) // a dense vector containing the mean valu
e for each column
println(summary.variance) // column-wise variance
println(summary.numNonzeros) // number of nonzeros in each column
```

下面我们具体看看`colStats`方法的实现。

```
def colStats(X: RDD[Vector]): MultivariateStatisticalSummary = {
    new RowMatrix(X).computeColumnSummaryStatistics()
  }
```

上面的代码非常明显，利用传人的`RDD`创建`RowMatrix`对象，利用方法`computeColumnSummaryStatistics`统计指标。

```
def computeColumnSummaryStatistics(): MultivariateStatisticalSum
mary = {
    val summary = rows.treeAggregate(new MultivariateOnlineSumma
rizer)(
      (aggregator, data) => aggregator.add(data),
      (aggregator1, aggregator2) => aggregator1.merge(aggregator
2))
    updateNumRows(summary.count)
    summary
  }
```

上面的代码调用了 `RDD` 的 `treeAggregate` 方法， `treeAggregate` 是聚合方法，它迭代处理 `RDD` 中的数据，其中， `(aggregator, data) => aggregator.add(data)` 处理每条数据，将其添加到 `MultivariateOnlineSummarizer` ， `(aggregator1, aggregator2) => aggregator1.merge(aggregator2)` 将不同分区的 `MultivariateOnlineSummarizer` 对象汇总。所以上述代码实现的重点是 `add` 方法和 `merge` 方法。它们都定义在 `MultivariateOnlineSummarizer` 中。我们先来看 `add` 代码。

```
@Since("1.1.0")
  def add(sample: Vector): this.type = add(sample, 1.0)
  private[spark] def add(instance: Vector, weight: Double): this.
type = {
    if (weight == 0.0) return this
    if (n == 0) {
      n = instance.size
      currMean = Array.ofDim[Double](n)
      currM2n = Array.ofDim[Double](n)
      currM2 = Array.ofDim[Double](n)
      currL1 = Array.ofDim[Double](n)
      nnz = Array.ofDim[Double](n)
      currMax = Array.fill[Double](n)(Double.MinValue)
      currMin = Array.fill[Double](n)(Double.MaxValue)
    }
    val localCurrMean = currMean
    val localCurrM2n = currM2n
    val localCurrM2 = currM2
```

```
    val localCurrL1 = currL1
    val localNnz = nnz
    val localCurrMax = currMax
    val localCurrMin = currMin
    instance.foreachActive { (index, value) =>
      if (value != 0.0) {
        if (localCurrMax(index) < value) {
          localCurrMax(index) = value
        }
        if (localCurrMin(index) > value) {
          localCurrMin(index) = value
        }
        val prevMean = localCurrMean(index)
        val diff = value - prevMean
        localCurrMean(index) = prevMean + weight * diff / (local
Nnz(index) + weight)
        localCurrM2n(index) += weight * (value - localCurrMean(i
ndex)) * diff
        localCurrM2(index) += weight * value * value
        localCurrL1(index) += weight * math.abs(value)
        localNnz(index) += weight
      }
    }
    weightSum += weight
    weightSquareSum += weight * weight
    totalCnt += 1
    this
  }
```

这段代码使用了在线算法来计算均值和方差。根据文献【1】的介绍，计算均值和方差遵循如下的迭代公式：

$$\bar{x}_n = \frac{(n-1)\,\bar{x}_{n-1} + x_n}{n} = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n}$$

$$M_{2,n} = M_{2,n-1} + (x_n - \bar{x}_{n-1})(x_n - \bar{x}_n)$$

$$s_n^2 = \frac{M_{2,n}}{n-1}$$

$$\sigma_n^2 = \frac{M_{2,n}}{n}$$

在上面的公式中，`x` 表示样本均值，`s` 表示样本方差，`delta` 表示总体方差。`MLlib` 实现的是带有权重的计算，所以使用的迭代公式略有不同，参考文献【2】。

$$\bar{x}_n = \bar{x}_{n-1} + \frac{w_n(x_n - \bar{x}_{n-1})}{w_n + \sum_{j=1}^{n-1} w_j}$$

$$M_{2,n} = M_{2,n-1} + w_n * (x_n - \bar{x}_{n-1})(x_n - \bar{x}_n)$$

$$s_n^2 = \frac{M_{2,n}}{\sum_{j=1}^{n} w_j}$$

$$\sigma_n^2 = \frac{nM_{2,n}}{(n-1)\sum_{j=1}^{n} w_j}$$

`merge` 方法相对比较简单，它只是对两个 `MultivariateOnlineSummarizer` 对象的指标作合并操作。

```
def merge(other: MultivariateOnlineSummarizer): this.type = {
   if (this.weightSum != 0.0 && other.weightSum != 0.0) {
     totalCnt += other.totalCnt
     weightSum += other.weightSum
     weightSquareSum += other.weightSquareSum
     var i = 0
     while (i < n) {
       val thisNnz = nnz(i)
       val otherNnz = other.nnz(i)
       val totalNnz = thisNnz + otherNnz
       if (totalNnz != 0.0) {
         val deltaMean = other.currMean(i) - currMean(i)
         // merge mean together
         currMean(i) += deltaMean * otherNnz / totalNnz
         // merge m2n together，不单纯是累加
         currM2n(i) += other.currM2n(i) + deltaMean * deltaMean
* thisNnz * otherNnz / totalNnz
         // merge m2 together
         currM2(i) += other.currM2(i)
         // merge l1 together
         currL1(i) += other.currL1(i)
         // merge max and min
```

```
          currMax(i) = math.max(currMax(i), other.currMax(i))
          currMin(i) = math.min(currMin(i), other.currMin(i))
        }
        nnz(i) = totalNnz
        i += 1
      }
    } else if (weightSum == 0.0 && other.weightSum != 0.0) {
      this.n = other.n
      this.currMean = other.currMean.clone()
      this.currM2n = other.currM2n.clone()
      this.currM2 = other.currM2.clone()
      this.currL1 = other.currL1.clone()
      this.totalCnt = other.totalCnt
      this.weightSum = other.weightSum
      this.weightSquareSum = other.weightSquareSum
      this.nnz = other.nnz.clone()
      this.currMax = other.currMax.clone()
      this.currMin = other.currMin.clone()
    }
    this
  }
```

这里需要注意的是，在线算法的并行化实现是一种特殊情况。例如样本集 `X` 分到两个不同的分区，分别为 `X_A` 和 `X_B` ，那么它们的合并需要满足下面的公式：

$$\delta = \bar{x}_B - \bar{x}_A$$
$$\bar{x}_X = \bar{x}_A + \delta \cdot \frac{n_B}{n_X}$$
$$M_{2,X} = M_{2,A} + M_{2,B} + \delta^2 \cdot \frac{n_A n_B}{n_X}.$$

依靠文献【3】我们可以知道，样本方差的无偏估计由下面的公式给出：

$$\mu^* = \frac{\sum_{i=1}^N w_i \mathbf{x}_i}{\sum_{i=1}^N w_i}.$$

$$\Sigma = \frac{\sum_{i=1}^N w_i}{\left(\sum_{i=1}^N w_i\right)^2 - \sum_{i=1}^N w_i^2} \sum_{i=1}^N w_i \left(\mathbf{x}_i - \mu^*\right)^T \left(\mathbf{x}_i - \mu^*\right).$$

所以，真实的样本均值和样本方差通过下面的代码实现。

```
override def mean: Vector = {
    val realMean = Array.ofDim[Double](n)
    var i = 0
    while (i < n) {
      realMean(i) = currMean(i) * (nnz(i) / weightSum)
      i += 1
    }
    Vectors.dense(realMean)
  }
 override def variance: Vector = {
    val realVariance = Array.ofDim[Double](n)
    val denominator = weightSum - (weightSquareSum / weightSum)
    // Sample variance is computed, if the denominator is less t
han 0, the variance is just 0.
    if (denominator > 0.0) {
      val deltaMean = currMean
      var i = 0
      val len = currM2n.length
      while (i < len) {
        realVariance(i) = (currM2n(i) + deltaMean(i) * deltaMean
(i) * nnz(i) *
          (weightSum - nnz(i)) / weightSum) / denominator
        i += 1
      }
    }
    Vectors.dense(realVariance)
  }
```

参考文献

【1】Algorithms for calculating variance

【2】Updating mean and variance estimates: an improved method

【3】Weighted arithmetic mean

相关性系数

计算两个数据集的相关性是统计中的常用操作。在 `MLlib` 中提供了计算多个数据集两两相关的方法。目前支持的相关性方法有皮尔森(`Pearson`)相关和斯皮尔曼(`Spearman`)相关。

`Statistics` 提供方法计算数据集的相关性。根据输入的类型，两个 `RDD[Double]` 或者一个 `RDD[Vector]` ，输出将会是一个 `Double` 值或者相关性矩阵。下面是一个应用的例子。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.stat.Statistics
val sc: SparkContext = ...
val seriesX: RDD[Double] = ... // a series
val seriesY: RDD[Double] = ... // must have the same number of partitions and cardinality as seriesX
// compute the correlation using Pearson's method. Enter "spearman" for Spearman's method. If a
// method is not specified, Pearson's method will be used by default.
val correlation: Double = Statistics.corr(seriesX, seriesY, "pearson")
val data: RDD[Vector] = ... // note that each Vector is a row and not a column
// calculate the correlation matrix using Pearson's method. Use "spearman" for Spearman's method.
// If a method is not specified, Pearson's method will be used by default.
val correlMatrix: Matrix = Statistics.corr(data, "pearson")
```

这个例子中我们看到，计算相关性的入口函数是 `Statistics.corr` ，当输入的数据集是两个 `RDD[Double]` 时，它的实际实现是 `Correlations.corr` ，当输入数据集是 `RDD[Vector]` 时，它的实际实现是 `Correlations.corrMatrix` 。下文会分别分析这两个函数。

```
def corr(x: RDD[Double],
       y: RDD[Double],
       method: String = CorrelationNames.defaultCorrName): Double
 = {
    val correlation = getCorrelationFromName(method)
    correlation.computeCorrelation(x, y)
  }
def corrMatrix(X: RDD[Vector],
      method: String = CorrelationNames.defaultCorrName): Matrix
 = {
    val correlation = getCorrelationFromName(method)
    correlation.computeCorrelationMatrix(X)
  }
```

这两个函数的第一步就是获得对应方法名的相关性方法的实现对象。并且如果输入数据集是两个 `RDD[Double]` ，`MLlib` 会将其统一转换为 `RDD[Vector]` 进行处理。

```
def computeCorrelationWithMatrixImpl(x: RDD[Double], y: RDD[Doub
le]): Double = {
    val mat: RDD[Vector] = x.zip(y).map { case (xi, yi) => new D
enseVector(Array(xi, yi)) }
    computeCorrelationMatrix(mat)(0, 1)
  }
```

不同的相关性方法，`computeCorrelationMatrix` 的实现不同。下面分别介绍皮尔森相关与斯皮尔曼相关的实现。

1 皮尔森相关系数

皮尔森相关系数也叫皮尔森积差相关系数，是用来反映两个变量相似程度的统计量。或者说可以用来计算两个向量的相似度（在基于向量空间模型的文本分类、用户喜好推荐系统中都有应用）。皮尔森相关系数计算公式如下：

$$\rho_{X,Y} = \frac{COV(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(X-\mu_Y)]}{\sigma_X \sigma_Y} = \frac{E(XY)-E(X)E(Y)}{\sqrt{E[X^2]-E[X]^2}\sqrt{E[Y^2]-E[Y]^2}}$$

当两个变量的线性关系增强时，相关系数趋于1或-1。正相关时趋于1，负相关时趋于-1。当两个变量独立时相关系统为0，但反之不成立。当 `Y` 和 `X` 服从联合正态分布时，其相互独立和不相关是等价的。 皮尔森相关系数的计算通过下面代码实现。

```
override def computeCorrelationMatrix(X: RDD[Vector]): Matrix =
{
    val rowMatrix = new RowMatrix(X)
    //计算协方差矩阵
    val cov = rowMatrix.computeCovariance()
    computeCorrelationMatrixFromCovariance(cov)
  }
def computeCorrelationMatrixFromCovariance(covarianceMatrix: Mat
rix): Matrix = {
    val cov = covarianceMatrix.toBreeze.asInstanceOf[BDM[Double]
]
    val n = cov.cols
    // 计算对角元素的标准差
    var i = 0
    while (i < n) {
      cov(i, i) = if (closeToZero(cov(i, i))) 0.0 else math.sqrt
(cov(i, i))
      i +=1
    }
    // Loop through columns since cov is column major
    var j = 0
    var sigma = 0.0
    var containNaN = false
    while (j < n) {
      sigma = cov(j, j)
      i = 0
      while (i < j) {
        val corr = if (sigma == 0.0 || cov(i, i) == 0.0) {
          containNaN = true
          Double.NaN
        } else {
          //根据上文的公式计算，即cov(x,y)/(sigma_x * sigma_y)
          cov(i, j) / (sigma * cov(i, i))
        }
```

```
        cov(i, j) = corr
        cov(j, i) = corr
        i += 1
      }
      j += 1
    }
    // put 1.0 on the diagonals
    i = 0
    while (i < n) {
      cov(i, i) = 1.0
      i +=1
    }
    Matrices.fromBreeze(cov)
  }
```

2 斯皮尔曼相关系数

使用皮尔森线性相关系数有2个局限：首先，必须假设数据是成对地从正态分布中取得的；其次，数据至少在逻辑范围内是等距的。对不服从正态分布的资料不符合使用矩相关系数来描述关联性。此时可采用秩相关（ `rank correlation` ），也称等级相关，来描述两个变量之间的关联程度与方向。斯皮尔曼秩相关系数就是其中一种。

斯皮尔曼秩相关系数定义为排序变量(`ranked variables`)之间的皮尔逊相关系数。对于大小为 `n` 的样本集，将原始的数据 `X_i` 和 `Y_i` 转换成排序变量 `rgX_i` 和 `rgY_i` ，然后按照皮尔森相关系数的计算公式进行计算。

$$r_s = \rho_{\mathrm{rg}_X,\mathrm{rg}_Y} = \frac{\mathrm{cov}(\mathrm{rg}_X,\mathrm{rg}_Y)}{\sigma_{\mathrm{rg}_X}\sigma_{\mathrm{rg}_Y}}$$

下面的代码将原始数据转换成了排序数据。

```
override def computeCorrelationMatrix(X: RDD[Vector]): Matrix =
{
    // ((columnIndex, value), rowUid)
    //使用zipWithUniqueId产生的rowUid全局唯一
    val colBased = X.zipWithUniqueId().flatMap { case (vec, uid)
 =>
      vec.toArray.view.zipWithIndex.map { case (v, j) =>
```

```
        ((j, v), uid)
      }
    }
    // 通过(columnIndex, value)全局排序，排序的好处是使下面只需迭代一次
    val sorted = colBased.sortByKey()
    // 分配全局的ranks (using average ranks for tied values)
    val globalRanks = sorted.zipWithIndex().mapPartitions { iter
 =>
      var preCol = -1
      var preVal = Double.NaN
      var startRank = -1.0
      var cachedUids = ArrayBuffer.empty[Long]
      val flush: () => Iterable[(Long, (Int, Double))] = () => {
        val averageRank = startRank + (cachedUids.size - 1) / 2.0

        val output = cachedUids.map { uid =>
          (uid, (preCol, averageRank))
        }
        cachedUids.clear()
        output
      }
      iter.flatMap { case (((j, v), uid), rank) =>
        // 如果有新的值或者cachedUids过大, 调用flush
        if (j != preCol || v != preVal || cachedUids.size >= 100
00000) {
          val output = flush()
          preCol = j
          preVal = v
          startRank = rank
          cachedUids += uid
          output
        } else {
          cachedUids += uid
          Iterator.empty
        }
      } ++ flush()
    }
   //使用rank值代替原来的值
   val groupedRanks = globalRanks.groupByKey().map { case (uid,
iter) =>
```

```
        // 根据列索引排序
        Vectors.dense(iter.toSeq.sortBy(_._1).map(_._2).toArray)
    }
```

在每个分区内部，对于列索引相同且值相同的数据对，我们为其分配平均 `rank` 值。平均 `rank` 的计算方式如下面公式所示：

$$\mathrm{rank}_{avg} = rank_{start} + \frac{n-1}{2}$$

其中 `rank_start` 表示列索引相同且值相同的数据对在分区中第一次出现时的索引位置， `n` 表示列索引相同且值相同的数据对出现的次数。

3 参考文献

【1】Pearson product-moment correlation coefficient

【2】Spearman's rank correlation coefficient

【3】相关性检验--Spearman秩相关系数和皮尔森相关系数

分层取样

先将总体的单位按某种特征分为若干次级总体（层），然后再从每一层内进行单纯随机抽样，组成一个样本的统计学计算方法叫做分层抽样。
在 `spark.mllib` 中，用 `key` 来分层。

与存在于 `spark.mllib` 中的其它统计函数不同，分层采样方法 `sampleByKey` 和 `sampleByKeyExact` 可以在 `key-value` 对的 `RDD` 上执行。在分层采样中，可以认为 `key` 是一个标签， `value` 是特定的属性。例如， `key` 可以是男人或者女人或者文档 `id`,它相应的 `value` 可能是一组年龄或者是文档中的词。 `sampleByKey` 方法通过掷硬币的方式决定是否采样一个观察数据， 因此它需要我们传递（ `pass over` ）数据并且提供期望的数据大小(`size`)。 `sampleByKeyExact` 比每层使用 `sampleByKey` 随机抽样需要更多的有意义的资源，但是它能使样本大小的准确性达到了 `99.99%` 。

sampleByKeyExact()允许用户准确抽取 `f_k * n_k` 个样本， 这里 `f_k` 表示期望获取键为 `k` 的样本的比例， `n_k` 表示键为 `k` 的键值对的数量。下面是一个使用的例子：

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.rdd.PairRDDFunctions
val sc: SparkContext = ...
val data = ... // an RDD[(K, V)] of any key value pairs
val fractions: Map[K, Double] = ... // specify the exact fractio
n desired from each key
// Get an exact sample from each stratum
val approxSample = data.sampleByKey(withReplacement = false, fra
ctions)
val exactSample = data.sampleByKeyExact(withReplacement = false,
 fractions)
```

当 `withReplacement` 为 `true` 时，采用 `PoissonSampler` 取样器，当 `withReplacement` 为 `false` 使，采用 `BernoulliSampler` 取样器。

```
def sampleByKey(withReplacement: Boolean,
      fractions: Map[K, Double],
      seed: Long = Utils.random.nextLong): RDD[(K, V)] = self.wi
thScope {
    val samplingFunc = if (withReplacement) {
      StratifiedSamplingUtils.getPoissonSamplingFunction(self, f
ractions, false, seed)
    } else {
      StratifiedSamplingUtils.getBernoulliSamplingFunction(self,
 fractions, false, seed)
    }
    self.mapPartitionsWithIndex(samplingFunc, preservesPartition
ing = true)
  }
def sampleByKeyExact(
      withReplacement: Boolean,
      fractions: Map[K, Double],
      seed: Long = Utils.random.nextLong): RDD[(K, V)] = self.wi
thScope {
    val samplingFunc = if (withReplacement) {
      StratifiedSamplingUtils.getPoissonSamplingFunction(self, f
ractions, true, seed)
    } else {
      StratifiedSamplingUtils.getBernoulliSamplingFunction(self,
 fractions, true, seed)
    }
    self.mapPartitionsWithIndex(samplingFunc, preservesPartition
ing = true)
  }
```

下面我们分别来看 `sampleByKey` 和 `sampleByKeyExact` 的实现。

1 `sampleByKey` 的实现

当我们需要不重复抽样时，我们需要用泊松抽样器来抽样。当需要重复抽样时，用伯努利抽样器抽样。`sampleByKey` 的实现比较简单，它就是统一的随机抽样。

1.1 泊松抽样器

我们首先看泊松抽样器的实现。

```
def getPoissonSamplingFunction[K: ClassTag, V: ClassTag](rdd: RDD
[(K, V)],
      fractions: Map[K, Double],
      exact: Boolean,
      seed: Long): (Int, Iterator[(K, V)]) => Iterator[(K, V)] =
 {
      (idx: Int, iter: Iterator[(K, V)]) => {
              //初始化随机生成器
              val rng = new RandomDataGenerator()
              rng.reSeed(seed + idx)
              iter.flatMap { item =>
                //获得下一个泊松值
                val count = rng.nextPoisson(fractions(item._1))
                if (count == 0) {
                  Iterator.empty
                } else {
                  Iterator.fill(count)(item)
                }
              }
            }
}
```

`getPoissonSamplingFunction` 返回的是一个函数，传递给 `mapPartitionsWithIndex` 处理每个分区的数据。这里 `RandomDataGenerator` 是一个随机生成器，它用于同时生成均匀值(`uniform values`)和泊松值(`Poisson values`)。

1.2 伯努利抽样器

```
def getBernoulliSamplingFunction[K, V](rdd: RDD[(K, V)],
      fractions: Map[K, Double],
      exact: Boolean,
      seed: Long): (Int, Iterator[(K, V)]) => Iterator[(K, V)] =
 {
    var samplingRateByKey = fractions
    (idx: Int, iter: Iterator[(K, V)]) => {
      //初始化随机生成器
      val rng = new RandomDataGenerator()
      rng.reSeed(seed + idx)
      // Must use the same invoke pattern on the rng as in getSe
qOp for without replacement
      // in order to generate the same sequence of random number
s when creating the sample
      iter.filter(t => rng.nextUniform() < samplingRateByKey(t._
1))
    }
  }
```

2 `sampleByKeyExact` 的实现

`sampleByKeyExact` 获取更准确的抽样结果，它的实现也分为两种情况，重复抽样和不重复抽样。前者使用泊松抽样器，后者使用伯努利抽样器。

2.1 泊松抽样器

```
val counts = Some(rdd.countByKey())
//计算立即接受的样本数量，并且为每层生成候选名单
val finalResult = getAcceptanceResults(rdd, true, fractions, cou
nts, seed)
//决定接受样本的阈值，生成准确的样本大小
val thresholdByKey = computeThresholdByKey(finalResult, fraction
s)
(idx: Int, iter: Iterator[(K, V)]) => {
     val rng = new RandomDataGenerator()
     rng.reSeed(seed + idx)
     iter.flatMap { item =>
          val key = item._1
          val acceptBound = finalResult(key).acceptBound
          // Must use the same invoke pattern on the rng as in g
etSeqOp for with replacement
          // in order to generate the same sequence of random nu
mbers when creating the sample
          val copiesAccepted = if (acceptBound == 0) 0L else rng
.nextPoisson(acceptBound)
          //候选名单
          val copiesWaitlisted = rng.nextPoisson(finalResult(key
).waitListBound)
          val copiesInSample = copiesAccepted +
            (0 until copiesWaitlisted).count(i => rng.nextUnifor
m() < thresholdByKey(key))
          if (copiesInSample > 0) {
            Iterator.fill(copiesInSample.toInt)(item)
          } else {
            Iterator.empty
          }
     }
}
```

2.2 伯努利抽样

```
def getBernoulliSamplingFunction[K, V](rdd: RDD[(K, V)],
      fractions: Map[K, Double],
      exact: Boolean,
      seed: Long): (Int, Iterator[(K, V)]) => Iterator[(K, V)] =
 {
    var samplingRateByKey = fractions
    //计算立即接受的样本数量，并且为每层生成候选名单
    val finalResult = getAcceptanceResults(rdd, false, fractions
, None, seed)
    //决定接受样本的阈值，生成准确的样本大小
    samplingRateByKey = computeThresholdByKey(finalResult, fract
ions)
    (idx: Int, iter: Iterator[(K, V)]) => {
      val rng = new RandomDataGenerator()
      rng.reSeed(seed + idx)
      // Must use the same invoke pattern on the rng as in getSe
qOp for without replacement
      // in order to generate the same sequence of random number
s when creating the sample
      iter.filter(t => rng.nextUniform() < samplingRateByKey(t._
1))
    }
  }
```

假设检测

假设检测是统计中有力的工具，它用于判断一个结果是否在统计上是显著的、这个结果是否有机会发生。`spark.mllib`目前支持皮尔森卡方检测。输入属性的类型决定是作拟合优度(`goodness of fit`)检测还是独立性检测。 拟合优度检测需要输入数据的类型是`vector`，独立性检测需要输入数据的类型是`Matrix`。

`spark.mllib`也支持输入数据类型为`RDD[LabeledPoint]`，它用来通过卡方独立性检测作特征选择。`Statistics`提供方法用来作皮尔森卡方检测。下面是一个例子：

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.stat.Statistics._
val sc: SparkContext = ...
val vec: Vector = ... // a vector composed of the frequencies of
 events
// 作皮尔森拟合优度检测
val goodnessOfFitTestResult = Statistics.chiSqTest(vec)
println(goodnessOfFitTestResult)
val mat: Matrix = ... // a contingency matrix
// 作皮尔森独立性检测
val independenceTestResult = Statistics.chiSqTest(mat)
println(independenceTestResult) // summary of the test including
 the p-value, degrees of freedom...
val obs: RDD[LabeledPoint] = ... // (feature, label) pairs.
// 独立性检测用于特征选择
val featureTestResults: Array[ChiSqTestResult] = Statistics.chiS
qTest(obs)
var i = 1
featureTestResults.foreach { result =>
    println(s"Column $i:\n$result")
    i += 1
}
```

另外，`spark.mllib` 提供了一个 `Kolmogorov-Smirnov (KS)` 检测的 `1-sample, 2-sided` 实现，用来检测概率分布的相等性。通过提供理论分布（现在仅仅支持正太分布）的名字以及它相应的参数， 或者提供一个计算累积分布(`cumulative distribution`)的函数，用户可以检测原假设或零假设(`null hypothesis`)：即样本是否来自于这个分布。用户检测正太分布，但是不提供分布参数，检测会默认该分布为标准正太分布。

`Statistics` 提供了一个运行 `1-sample, 2-sided KS` 检测的方法，下面就是一个应用的例子。

```
import org.apache.spark.mllib.stat.Statistics
val data: RDD[Double] = ... // an RDD of sample data
// run a KS test for the sample versus a standard normal distribution
val testResult = Statistics.kolmogorovSmirnovTest(data, "norm", 0, 1)
println(testResult)
// perform a KS test using a cumulative distribution function of our making
val myCDF: Double => Double = ...
val testResult2 = Statistics.kolmogorovSmirnovTest(data, myCDF)
```

流式显著性检测

显著性检验即用于实验处理组与对照组或两种不同处理的效应之间是否有差异，以及这种差异是否显著的方法。

常把一个要检验的假设记作 `H0` ,称为原假设（或零假设） (`null hypothesis` ），与 `H0` 对立的假设记作 `H1` ，称为备择假设(`alternative hypothesis` ）。

- 在原假设为真时，决定放弃原假设，称为第一类错误，其出现的概率通常记作 `alpha`
- 在原假设不真时，决定接受原假设，称为第二类错误，其出现的概率通常记作 `beta`

通常只限定犯第一类错误的最大概率 `alpha` ，不考虑犯第二类错误的概率 `beta` 。这样的假设检验又称为显著性检验，概率 `alpha` 称为显著性水平。

`MLlib` 提供一些检测的在线实现，用于支持诸如 `A/B` 测试的场景。这些检测可能执行在 `Spark Streaming` 的 `DStream[(Boolean,Double)]` 上，元组的第一个元素表示控制组(`control group (false)`)或者处理组(`treatment group (true)`), 第二个元素表示观察者的值。

流式显著性检测支持下面的参数：

- `peacePeriod` ：来自流中忽略的初始数据点的数量，用于减少 `novelty effects` ；
- `windowSize` ：执行假设检测的以往批次的数量。如果设置为0，将对之前所有的批次数据作累积处理。

`StreamingTest` 支持流式假设检测。下面是一个应用的例子。

```
val data = ssc.textFileStream(dataDir).map(line => line.split(","
) match {
  case Array(label, value) => BinarySample(label.toBoolean, valu
e.toDouble)
})
val streamingTest = new StreamingTest()
  .setPeacePeriod(0)
  .setWindowSize(0)
  .setTestMethod("welch")
val out = streamingTest.registerStream(data)
out.print()
```

参考文献

【1】显著性检验

随机数生成

随机数生成在随机算法、性能测试中非常有用， `spark.mllib` 支持生成随机的 `RDD` , `RDD` 的独立同分布(`iid`)的值来自于给定的分布：均匀分布、标准正太分布、泊松分布。

`RandomRDDs` 提供工厂方法生成随机的双精度 `RDD` 或者向量 `RDD` 。下面的例子生成了一个随机的双精度 `RDD` ，它的值来自于标准的正太分布 `N(0,1)` 。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.random.RandomRDDs._
val sc: SparkContext = ...
// Generate a random double RDD that contains 1 million i.i.d. values drawn from the
// standard normal distribution `N(0, 1)`, evenly distributed in 10 partitions.
val u = normalRDD(sc, 1000000L, 10)
// Apply a transform to get a random double RDD following `N(1, 4)`.
val v = u.map(x => 1.0 + 2.0 * x)
```

`normalRDD` 的实现如下面代码所示。

```
def normalRDD(
      sc: SparkContext,
      size: Long,
      numPartitions: Int = 0,
      seed: Long = Utils.random.nextLong()): RDD[Double] = {
    val normal = new StandardNormalGenerator()
    randomRDD(sc, normal, size, numPartitionsOrDefault(sc, numPa
rtitions), seed)
  }
def randomRDD[T: ClassTag](
      sc: SparkContext,
      generator: RandomDataGenerator[T],
      size: Long,
      numPartitions: Int = 0,
      seed: Long = Utils.random.nextLong()): RDD[T] = {
    new RandomRDD[T](sc, size, numPartitionsOrDefault(sc, numPar
titions), generator, seed)
  }
private[mllib] class RandomRDD[T: ClassTag](sc: SparkContext,
    size: Long,
    numPartitions: Int,
    @transient private val rng: RandomDataGenerator[T],
    @transient private val seed: Long = Utils.random.nextLong) e
xtends RDD[T](sc, Nil)
```

核密度估计

1 理论分析

核密度估计是在概率论中用来估计未知的密度函数，属于非参数检验方法之一。假设我们有 `n` 个数$x{1},x{2},...,x_{n}$，要计算某个数 `X` 的概率密度有多大， 可以通过下面的核密度估计方法估计。

$$f(x) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right)$$

在上面的式子中， `K` 为核密度函数， `h` 为窗宽。核密度函数的原理比较简单，在我们知道某一事物的概率分布的情况下，如果某一个数在观察中出现了，我们可以认为这个数的概率密度很大，和这个数比较近的数的概率密度也会比较大，而那些离这个数远的数的概率密度会比较小。

基于这种想法，针对观察中的第一个数，我们可以用 `K` 去拟合我们想象中的那个远小近大概率密度。对每一个观察数拟合出的多个概率密度分布函数，取平均。如果某些数是比较重要的，则可以取加权平均。需要说明的一点是，核密度的估计并不是找到真正的分布函数。

在 `MLlib` 中，仅仅支持以高斯核做核密度估计。以高斯核做核密度估计时核密度估计公式（**1**）如下：

$$p_n(x) = \frac{1}{\sqrt{2\pi}nh_n}\sum_{j=1}^{n} e^{-\frac{(x-x_j)^2}{2h_n^2}}$$

2 例子

`KernelDensity` 提供了方法通过样本 `RDD` 计算核密度估计。下面的例子给出了使用方法。

```
import org.apache.spark.mllib.stat.KernelDensity
import org.apache.spark.rdd.RDD
val data: RDD[Double] = ... // an RDD of sample data
// Construct the density estimator with the sample data and a st
andard deviation for the Gaussian
// kernels
val kd = new KernelDensity()
  .setSample(data)
  .setBandwidth(3.0)
// Find density estimates for the given values
val densities = kd.estimate(Array(-1.0, 2.0, 5.0))
```

3 代码实现

通过调用 `KernelDensity.estimate` 方法来实现核密度函数估计。看下面的代码。

```
def estimate(points: Array[Double]): Array[Double] = {
    val sample = this.sample
    val bandwidth = this.bandwidth
    val n = points.length
    // 在每个高斯密度函数计算中，这个值都需要计算，所以提前计算。
    val logStandardDeviationPlusHalfLog2Pi = math.log(bandwidth)
 + 0.5 * math.log(2 * math.Pi)
    val (densities, count) = sample.aggregate((new Array[Double]
(n), 0L))(
      (x, y) => {
        var i = 0
        while (i < n) {
          x._1(i) += normPdf(y, bandwidth, logStandardDeviationP
lusHalfLog2Pi, points(i))
          i += 1
        }
        (x._1, x._2 + 1)
      },
      (x, y) => {
        //daxpy函数的作用是将一个向量加上另一个向量的值，即：dy[i]+=da*d
x[i]，其中da为常数
        blas.daxpy(n, 1.0, y._1, 1, x._1, 1)
        (x._1, x._2 + y._2)
      })
    //在向量上乘一个常数
    blas.dscal(n, 1.0 / count, densities, 1)
    densities
  }
}
```

上述代码的 `seqOp` 函数中调用了 `normPdf` ，这个函数用于计算核函数为高斯分布的概率密度函数。参见上面的公式**(1)**。公式**(1)**的实现如下面代码。

```
def normPdf(
      mean: Double,
      standardDeviation: Double,
      logStandardDeviationPlusHalfLog2Pi: Double,
      x: Double): Double = {
    val x0 = x - mean
    val x1 = x0 / standardDeviation
    val logDensity = -0.5 * x1 * x1 - logStandardDeviationPlusHa
lfLog2Pi
    math.exp(logDensity)
  }
```

该方法首先将公式**(1)**取对数，计算结果，然后再对计算结果取指数。

参考文献

【1】核密度估计

【2】R语言与非参数统计（核密度估计）

交换最小二乘

1 什么是ALS

`ALS` 是交替最小二乘（ `alternating least squares` ）的简称。在机器学习中， `ALS` 特指使用交替最小二乘求解的一个协同推荐算法。它通过观察到的所有用户给商品的打分，来推断每个用户的喜好并向用户推荐适合的商品。举个例子，我们看下面一个 `8*8` 的用户打分矩阵。

			5				1
	2				3		
6		5					
				6			
		7				6	
				?		9	
			4		2		
9						3	

这个矩阵的每一行代表一个用户 `（u1,u2,…,u8）` 、每一列代表一个商品 `（v1,v2,…,v8）` 、用户的打分为 `1-9` 分。这个矩阵只显示了观察到的打分，我们需要推测没有观察到的打分。比如 `（u6，v5）` 打分多少？如果以数独的方式来解决这个问题，可以得到唯一的结果。 因为数独的规则很强，每添加一条规则，就让整个系统的自由度下降一个量级。当我们满足所有的规则时，整个系统的自由度就降为 `1` 了，也就得出了唯一的结果。对于上面的打分矩阵，如果我们不添加任何条件的话，也即打分之间是相互独立的，我们就没法得到 `（u6，v5）` 的打分。 所以在这个用户打分矩阵的基础上，我们需要提出一个限制其自由度的合理假设，使得我们可以通过观察已有打分来猜测未知打分。

`ALS` 的核心就是这样一个假设：打分矩阵是近似低秩的。换句话说，就是一个 `m*n` 的打分矩阵可以由分解的两个小矩阵 `U（m*k）` 和 `V（k*n）` 的乘积来近似，即$A=U{V}^{T},k <= m,n$。这就是 `ALS` 的矩阵分解方法。这样我们把系统的自由度从 `O(mn)` 降到了 `O((m+n)k)` 。

那么 `ALS` 的低秩假设为什么是合理的呢？我们描述一个人的喜好经常是在一个抽象的低维空间上进行的，并不需要一一列出他喜好的事物。例如，我喜好看侦探影片，可能代表我喜欢《神探夏洛特》、《神探狄仁杰》等。这些影片都符合我对自己喜好的描述，也就是说他们在这个抽象的低维空间的投影和我的喜好相似。再抽象一些来描述这个问题，我们把某个人的喜好映射到了低维向量 `ui` 上，同时将某个影片的特征映射到了维度相同的向量 `vj` 上，那么这个人和这个影片的相似度就可以表述成这两个向量之间的内积$u*{i}^{T}*v{j}$ 。 我们把打分理解成相似度，那么打分矩阵A就可以由用户喜好矩阵和产品特征矩阵的乘积$ U{V}^{T} $来近似了。

低维空间的选取是一个问题。这个低维空间要能够很好的区分事物，那么就需要一个明确的可量化目标，这就是重构误差。在 `ALS` 中我们使用F范数来量化重构误差，就是每个元素重构误差的平方和。这里存在一个问题，我们只观察到部分打分， `A` 中的大量未知元是我们想推断的，所以这个重构误差是包含未知数的。 解决方案很简单：只计算已知打分的重构误差。

$$\sum_{i,\mathrm{j}\in\mathrm{R}}(a_{ij}-u_i v_j^T)^2$$

后面的章节我们将从原理上讲解 `spark` 中实现的 `ALS` 模型。

2 spark中ALS的实现原理

`Spark` 利用交换最小二乘解决矩阵分解问题分两种情况：数据集是显式反馈和数据集是隐式反馈。由于隐式反馈算法的原理是在显示反馈算法原理的基础上作的修改，所以我们在此只会具体讲解数据集为隐式反馈的算法。 算法实现所依据的文献见参考文献【1】。

2.1 介绍

从广义上讲，推荐系统基于两种不同的策略：基于内容的方法和基于协同过滤的方法。 `Spark` 中使用协同过滤的方式。协同过滤分析用户以及用户相关的产品的相关性，用以识别新的用户-产品相关性。协同过滤系统需要的唯一信息是用户过去的行为信息，比如对产品的评价信息。协同过滤是领域无关的，所以它可以方便解决基于内容方法难以解决的许多问题。

推荐系统依赖不同类型的输入数据，最方便的是高质量的显式反馈数据，它们包含用户对感兴趣商品明确的评价。例如， `Netflix` 收集的用户对电影评价的星星等级数据。但是显式反馈数据不一定总是找得到，因此推荐系统可以从更丰富的

隐式反馈信息中推测用户的偏好。 隐式反馈类型包括购买历史、浏览历史、搜索模式甚至鼠标动作。例如，购买同一个作者许多书的用户可能喜欢这个作者。

许多研究都集中在处理显式反馈，然而在很多应用场景下，应用程序重点关注隐式反馈数据。因为可能用户不愿意评价商品或者由于系统限制我们不能收集显式反馈数据。在隐式模型中，一旦用户允许收集可用的数据，在客户端并不需要额外的显式数据。文献中的系统避免主动地向用户收集显式反馈信息，所以系统仅仅依靠隐式信息。

了解隐式反馈的特点非常重要，因为这些特质使我们避免了直接调用基于显式反馈的算法。最主要的特点有如下几种：

- （1） 没有负反馈。通过观察用户行为，我们可以推测那个商品他可能喜欢，然后购买，但是我们很难推测哪个商品用户不喜欢。这在显式反馈算法中并不存在，因为用户明确告诉了我们哪些他喜欢哪些他不喜欢。
- （2） 隐式反馈是内在的噪音。虽然我们拼命的追踪用户行为，但是我们仅仅只是猜测他们的偏好和真实动机。例如，我们可能知道一个人的购买行为，但是这并不能完全说明偏好和动机，因为这个商品可能作为礼物被购买而用户并不喜欢它。
- （3） 显示反馈的数值值表示偏好（ `preference` ），隐式回馈的数值值表示信任（ `confidence` ）。基于显示反馈的系统用星星等级让用户表达他们的喜好程度，例如一颗星表示很不喜欢，五颗星表示非常喜欢。基于隐式反馈的数值值描述的是动作的频率，例如用户购买特定商品的次数。一个较大的值并不能表明更多的偏爱。但是这个值是有用的，它描述了在一个特定观察中的信任度。 一个发生一次的事件可能对用户偏爱没有用，但是一个周期性事件更可能反映一个用户的选择。
- （4） 评价隐式反馈推荐系统需要合适的手段。

2.2 显式反馈模型

潜在因素模型由一个针对协同过滤的交替方法组成，它以一个更加全面的方式发现潜在特征来解释观察的 `ratings` 数据。我们关注的模型由奇异值分解（ `SVD` ）推演而来。一个典型的模型将每个用户 `u` （包含一个用户-因素向量 `ui` ）和每个商品 `v` （包含一个用户-因素向量 `vj` ）联系起来。 预测通过内

积$r{ij}=u{i}^{T}v_{j}$来实现。另一个需要关注的地方是参数估计。许多当前的工作都应用到了显式反馈数据集中，这些模型仅仅基于观察到的 `rating` 数据直接建模，同时通过一个适当的正则化来避免过拟合。公式如下：

$$\min_{u,v}\sum_{r_{ij}}(r_{\mathrm{ij}}-u_i^T v_j)^2+\lambda(||u_i||^2+||v_j||^2) \qquad (2.1)$$

在公式(2.1)中， `lambda` 是正则化的参数。正规化是为了防止过拟合的情况发生，具体参见文献【3】。这样，我们用最小化重构误差来解决协同推荐问题。我们也成功将推荐问题转换为了最优化问题。

2.3 隐式反馈模型

在显式反馈的基础上，我们需要做一些改动得到我们的隐式反馈模型。首先，我们需要形式化由$r{ij}$变量衡量的信任度的概念。我们引入了一组二元变量$p{ij}$，它表示用户 `u` 对商品 `v` 的偏好。p_{ij}的公式如下：

$$p_{\mathrm{ij}}=\begin{cases}1, & r_{\mathrm{ij}}>0\\ 0, & r_{\mathrm{ij}}=0\end{cases} \qquad (2.2)$$

换句话说，如果用户购买了商品，我们认为用户喜欢该商品，否则我们认为用户不喜欢该商品。然而我们的信念（ `beliefs` ）与变化的信任（ `confidence` ）等级息息相关。首先，很自然的，$p{ij}$的值为0和低信任有关。用户对一个商品没有得到一个正的偏好可能源于多方面的原因，并不一定是不喜欢该商品。例如，用户可能并不知道该商品的存在。 另外，用户购买一个商品也并不一定是用户喜欢它。因此我们需要一个新的信任等级来显示用户偏爱某个商品。一般情况下，$r{ij}$越大，越能暗示用户喜欢某个商品。因此，我们引入了一组变量$c{ij}$，它衡量了我们观察到$p{ij}$的信任度。c_{ij}一个合理的选择如下所示：

$$c_{\mathrm{ij}}=1+\alpha r_{\mathrm{ij}} \qquad (2.3)$$

按照这种方式，我们存在最小限度的信任度，并且随着我们观察到的正偏向的证据越来越多，信任度也会越来越大。

我们的目的是找到用户向量 `ui` 以及商品向量 `vj` 来表明用户偏好。这些向量分别是用户因素（特征）向量和商品因素（特征）向量。本质上，这些向量将用户和商品映射到一个公用的隐式因素空间，从而使它们可以直接比较。这和用于显式数据集的矩阵分解技术类似，但是包含两点不一样的地方：（1）我们需要考虑不同的信任度，（2）最优化需要考虑所有可能的 `u，v` 对，而不仅仅是和观察数据相关的 `u，v` 对。因此，通过最小化下面的损失函数来计算相关因素（ `factors` ）。

$$\min_{u,v}\sum_{i,j}c_{ij}(p_{ij}-u_i^T v_j)^2+\lambda(\sum_u ||u_i||^2+\sum_i ||v_j||^2) \qquad (2.4)$$

2.4 求解最小化损失函数

考虑到损失函数包含 `m*n` 个元素， `m` 是用户的数量， `n` 是商品的数量。一般情况下， `m*n` 可以到达几百亿。这么多的元素应该避免使用随机梯度下降法来求解，因此，spark选择使用交替最优化方式求解。

公式（2.1）和公式（2.4）是非凸函数，无法求解最优解。但是，固定公式中的用户-特征向量或者商品-特征向量，公式就会变成二次方程，可以求出全局的极小值。交替最小二乘的计算过程是：交替的重新计算用户-特征向量和商品-特征向量，每一步都保证降低损失函数的值，直到找到极小值。 交替最小二乘法的处理过程如下所示：

（1） 初始化U^1和V^1

（2） 循环 k，k=1,2,...

$$U^{k+1}=\mathrm{argmin}_U f(U,V^k)$$
$$V^{k+1}=\mathrm{argmin}_V f(U^{k+1},V)$$

3 ALS在spark中的实现

在 `spark` 的源代码中， `ALS` 算法实现于 `org.apache.spark.ml.recommendation.ALS.scala` 文件中。我们以官方文档中的例子为起点，来分析 `ALS` 算法的分布式实现。下面是官方的例子：

```
//处理训练数据
val data = sc.textFile("data/mllib/als/test.data")
val ratings = data.map(_.split(',') match { case Array(user, ite
m, rate) =>
  Rating(user.toInt, item.toInt, rate.toDouble)
})
// 使用ALS训练推荐模型
val rank = 10
val numIterations = 10
val model = ALS.train(ratings, rank, numIterations, 0.01)
```

从代码中我们知道，训练模型用到了 `ALS.scala` 文件中的 `train` 方法，下面我们将详细介绍 `train` 方法的实现。在此之前，我们先了解一下 `train` 方法的参数表示的含义。

```
def train(
    ratings: RDD[Rating[ID]],  //训练数据
    rank: Int = 10,    //隐含特征数
    numUserBlocks: Int = 10, //分区数
    numItemBlocks: Int = 10,
    maxIter: Int = 10,    //迭代次数
    regParam: Double = 1.0,
    implicitPrefs: Boolean = false,
    alpha: Double = 1.0,
    nonnegative: Boolean = false,
    intermediateRDDStorageLevel: StorageLevel = StorageLevel.MEM
ORY_AND_DISK,
    finalRDDStorageLevel: StorageLevel = StorageLevel.MEMORY_AND
_DISK,
    checkpointInterval: Int = 10,
    seed: Long = 0L): MatrixFactorizationModel
```

以上定义中， `ratings` 指用户提供的训练数据，它包括用户 `id` 集、商品 `id` 集以及相应的打分集。 `rank` 表示隐含因素的数量，也即特征的数量。 `numUserBlocks` 和 `numItemBlocks` 分别指用户和商品的块数量，即分区数量。 `maxIter` 表示迭代次数。 `regParam` 表示最小二乘法中 `lambda` 值的大小。 `implicitPrefs` 表示我们的训练数据是否是隐式反馈数据。 `Nonnegative` 表示求解的最小二乘的值是否是非负,根据 `Nonnegative` 的值的不同， `spark` 使用了不同的求解方法。

下面我们分步骤分析 `train` 方法的处理流程。

- **(1)** 初始化 **`ALSPartitioner`** 和 **`LocalIndexEncoder`** 。

`ALSPartitioner` 实现了基于 `hash` 的分区，它根据用户或者商品 `id` 的 `hash` 值来进行分区。 `LocalIndexEncoder` 对 `（blockid，localindex）` 即 `（分区id，分区内索引）` 进行编码，并将其转换为一个整数，这个整数在高位存分区 `ID` ，在低位存对应分区的索引，在空间上尽量做到了不浪费。 同时也可以根据这个转换的整数分别获得 `blockid` 和 `localindex` 。这两个对象在后续的代码中会用到。

```
val userPart = new ALSPartitioner(numUserBlocks)
val itemPart = new ALSPartitioner(numItemBlocks)
val userLocalIndexEncoder = new LocalIndexEncoder(userPart.numPa
rtitions)
val itemLocalIndexEncoder = new LocalIndexEncoder(itemPart.numPa
rtitions)


//ALSPartitioner即HashPartitioner
class HashPartitioner(partitions: Int) extends Partitioner {
  def numPartitions: Int = partitions
  def getPartition(key: Any): Int = key match {
    case null => 0
    case _ => Utils.nonNegativeMod(key.hashCode, numPartitions)
  }
  override def equals(other: Any): Boolean = other match {
    case h: HashPartitioner =>
      h.numPartitions == numPartitions
    case _ =>
      false
  }
  override def hashCode: Int = numPartitions
}

//LocalIndexEncoder
private[recommendation] class LocalIndexEncoder(numBlocks: Int)
extends Serializable {

    private[this] final val numLocalIndexBits =
      math.min(java.lang.Integer.numberOfLeadingZeros(numBlocks
- 1), 31)
    //左移（<<,相当于乘2），右移（>>，相当于除2）和无符号右移（>>>，无符号
右移，忽略符号位，空位都以0补齐）
    private[this] final val localIndexMask = (1 << numLocalIndex
Bits) - 1
    //encodeIndex高位存分区ID，在低位存对应分区的索引
    def encode(blockId: Int, localIndex: Int): Int = {
      (blockId << numLocalIndexBits) | localIndex
    }
```

```
    @inline
    def blockId(encoded: Int): Int = {
      encoded >>> numLocalIndexBits
    }

    @inline
    def localIndex(encoded: Int): Int = {
      encoded & localIndexMask
    }
  }
```

- **(2)** 根据 `nonnegative` 参数选择解决矩阵分解的方法。

如果需要解的值为非负,即 `nonnegative` 为 `true` ，那么用非负最小二乘（ `NNLS` ）来解，如果没有这个限制，用乔里斯基（ `Cholesky` ）分解来解。

```
val solver = if (nonnegative) new NNLSSolver else new CholeskySo
lver
```

乔里斯基分解分解是把一个对称正定的矩阵表示成一个上三角矩阵 `U` 的转置和其本身的乘积的分解。在 `ml` 代码中，直接调用netlib-java封装的 `dppsv` 方法实现。

```
lapack.dppsv(“u”, k, 1, ne.ata, ne.atb, k, info)
```

可以深入 `dppsv` 代码（ `Fortran` 代码）了解更深的细节。我们分析的重点是非负正则化最小二乘的实现，因为在某些情况下，方程组的解为负数是没有意义的。虽然方程组可以得到精确解，但却不能取负值解。在这种情况下，其非负最小二乘解比方程的精确解更有意义。 `` `NNLS `` 在最优化模块会作详细讲解。

- **(3)** 将 `ratings` 数据转换为分区的格式。

将 `ratings` 数据转换为分区的形式，即 `（（用户分区id，商品分区id），分区数据集blocks））` 的形式，并缓存到内存中。其中分区id的计算是通过 `ALSPartitioner` 的 `getPartitions` 方法获得的，分区数据集由 `RatingBlock` 组成， 它表示 `（用户分区id，商品分区id ）` 对所对应的用户id集，商品id集，以及打分集，即 `（用户id集，商品id集，打分集）` 。

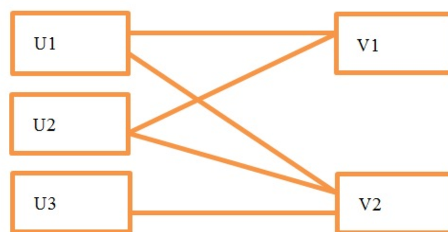
```
val blockRatings = partitionRatings(ratings, userPart, itemPart)
  .persist(intermediateRDDStorageLevel)

//以下是partitionRatings的实现
  //默认是10*10
  val numPartitions = srcPart.numPartitions * dstPart.numPartiti
ons
  ratings.mapPartitions { iter =>
      val builders = Array.fill(numPartitions)(new RatingBlockBu
ilder[ID])
      iter.flatMap { r =>
        val srcBlockId = srcPart.getPartition(r.user)
        val dstBlockId = dstPart.getPartition(r.item)
        //当前builder的索引位置
        val idx = srcBlockId + srcPart.numPartitions * dstBlockI
d
        val builder = builders(idx)
        builder.add(r)
        //如果某个builder的数量大于2048，那么构建一个分区
        if (builder.size >= 2048) { // 2048 * (3 * 4) = 24k
          builders(idx) = new RatingBlockBuilder
          //单元素集合
          Iterator.single(((srcBlockId, dstBlockId), builder.bui
ld()))
        } else {
          Iterator.empty
        }
      } ++ {
        builders.view.zipWithIndex.filter(_._1.size > 0).map { c
ase (block, idx) =>
          //用户分区id
          val srcBlockId = idx % srcPart.numPartitions
          //商品分区id
          val dstBlockId = idx / srcPart.numPartitions
          ((srcBlockId, dstBlockId), block.build())
        }
      }
    }.groupByKey().mapValues { blocks =>
      val builder = new RatingBlockBuilder[ID]
      blocks.foreach(builder.merge)
```

```
        builder.build()
      }.setName("ratingBlocks")
    }
```

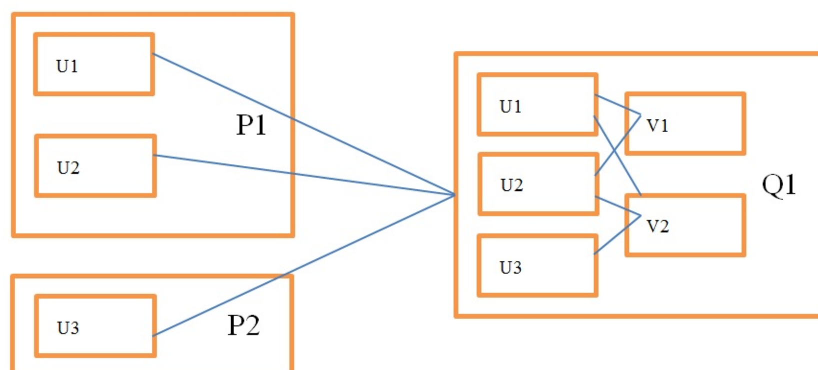
- （4）获取 `inblocks` 和 `outblocks` 数据。

获取 `inblocks` 和 `outblocks` 数据是数据处理的重点。我们知道，通信复杂度是分布式实现一个算法时要重点考虑的问题，不同的实现可能会对性能产生很大的影响。我们假设最坏的情况：即求解商品需要的所有用户特征都需要从其它节点获得。如下图3.1所示，求解 `v1` 需要获得 `u1` , `u2` ，求解 `v2` 需要获得 `u1` , `u2` , `u3` 等，在这种假设下，每步迭代所需的交换数据量是 `O(m*rank)` ，其中 `m` 表示所有观察到的打分集大小， `rank` 表示特征数量。



从图3.1中，我们知道，如果计算 `v1` 和 `v2` 是在同一个分区上进行的，那么我们只需要把 `u1` 和 `u2` 一次发给这个分区就好了，而不需要将 `u2` 分别发给 `v1` , `v2` ，这样就省掉了不必要的数据传输。

图3.2描述了如何在分区的情况下通过 `U` 来求解 `V` ，注意节点之间的数据交换量减少了。使用这种分区结构，我们需要在原始打分数据的基础上额外保存一些信息。



在 `Q1` 中，我们需要知道和 `v1` 相关联的用户向量及其对应的打分，从而构建最小二乘问题并求解。这部分数据不仅包含原始打分数据，还包含从每个用户分区收到的向量排序信息，在代码里称作 `InBlock` 。在 `P1` 中，我们要知道把 `u1` , `u2` 发给 `Q1` 。我们可以查看和 `u1` 相关联的所有产品来确定需要把 `u1` 发给谁，但每次迭代都扫一遍数据很不划算，所以在 `spark` 的实现中只计算一次这个信息，然后把结果通过 `RDD` 缓存起来重复使用。这部分数据我们在代码里称作 `OutBlock` 。 所以从 `U` 求解 `V` ，我们需要通过用户的 `OutBlock` 信息把用户向量发给商品分区，然后通过商品的 `InBlock` 信息构建最小二乘问题并求解。从 `V` 求解 `U` ，我们需要商品的 `OutBlock` 信息和用户的 `InBlock` 信息。所有的 `InBlock` 和 `OutBlock` 信息在迭代过程中都通过 `RDD` 缓存。打分数据在用户的 `InBlock` 和商品的 `InBlock` 各存了一份，但分区方式不同。这么做可以避免在迭代过程中原始数据的交换。

下面介绍获取 `InBlock` 和 `OutBlock` 的方法。下面的代码用来分别获取用户和商品的 `InBlock` 和 `OutBlock` 。

```
val (userInBlocks, userOutBlocks) =makeBlocks("user", blockRatin
gs,
 userPart, itemPart,intermediateRDDStorageLevel)
//交换userBlockId和itemBlockId以及其对应的数据
val swappedBlockRatings = blockRatings.map {
  case ((userBlockId, itemBlockId), RatingBlock(userIds, itemIds
, localRatings)) =>
    ((itemBlockId, userBlockId), RatingBlock(itemIds, userIds, l
ocalRatings))
}
val (itemInBlocks, itemOutBlocks) =makeBlocks("item", swappedBlo
ckRatings,
 itemPart, userPart,intermediateRDDStorageLevel)
```

我们会以求商品的 `InBlock` 以及用户的 `OutBlock` 为例来分析 `makeBlocks` 方法。因为在第（5）步中构建最小二乘的讲解中，我们会用到这两部分数据。

下面的代码用来求商品的 `InBlock` 信息。

```
val inBlocks = ratingBlocks.map {
  case ((srcBlockId, dstBlockId), RatingBlock(srcIds, dstIds, ra
```

```
tings)) =>
    val start = System.nanoTime()
    val dstIdSet = new OpenHashSet[ID](1 << 20)
    //将用户id保存到hashset中，用来去重
    dstIds.foreach(dstIdSet.add)
    val sortedDstIds = new Array[ID](dstIdSet.size)
    var i = 0
    var pos = dstIdSet.nextPos(0)
    while (pos != -1) {
      sortedDstIds(i) = dstIdSet.getValue(pos)
      pos = dstIdSet.nextPos(pos + 1)
      i += 1
    }
    //对用户id进行排序
    Sorting.quickSort(sortedDstIds)
    val dstIdToLocalIndex = new OpenHashMap[ID, Int](sortedDstId
s.length)
    i = 0
    while (i < sortedDstIds.length) {
      dstIdToLocalIndex.update(sortedDstIds(i), i)
      i += 1
    }
    //求取块内，用户id的本地位置
    val dstLocalIndices = dstIds.map(dstIdToLocalIndex.apply)
    //返回数据集
    (srcBlockId, (dstBlockId, srcIds, dstLocalIndices, ratings))
}.groupByKey(new ALSPartitioner(srcPart.numPartitions))
  .mapValues { iter =>
    val builder =
      new UncompressedInBlockBuilder[ID](new LocalIndexEncoder(d
stPart.numPartitions))
    iter.foreach { case (dstBlockId, srcIds, dstLocalIndices, ra
tings) =>
      builder.add(dstBlockId, srcIds, dstLocalIndices, ratings)
    }
    //构建非压缩块，并压缩为InBlock
    builder.build().compress()
  }.setName(prefix + "InBlocks")
  .persist(storageLevel)
```

这段代码首先对 `ratingBlocks` 数据集作 `map` 操作，将 `ratingBlocks` 转换成 `（商品分区id，（用户分区id，商品集合，用户id在分区中相对应的位置，打分）` 这样的集合形式。然后对这个数据集作 `groupByKey` 操作，以商品分区 `id` 为 `key` 值，处理 `key` 对应的值，将数据集转换成 `（商品分区id，InBlocks）` 的形式。 这里值得我们去分析的是输入块（ `InBlock` ）的结构。为简单起见，我们用图3.2为例来说明输入块的结构。

以 `Q1` 为例，我们需要知道关于 `v1` 和 `v2` 的所有打分： `(v1, u1, r11)，(v2, u1, r12)， (v1, u2, r21)， (v2, u2, r22)， (v2, u3, r32)` ，把这些项以 `Tuple` 的形式存储会存在问题，第一， `Tuple` 有额外开销，每个 `Tuple` 实例都需要一个指针，而每个 `Tuple` 所存的数据不过是两个ID和一个打分； 第二，存储大量的 `Tuple` 会降低垃圾回收的效率。所以 `spark` 实现中，是使用三个数组来存储打分的，如 `([v1, v2, v1, v2, v2], [u1, u1, u2, u2, u3], [r11, r12, r21, r22, r32])` 。这样不仅大幅减少了实例数量，还有效地利用了连续内存。

但是，光这么做并不够， `spark` 代码实现中，并没有存储用户的真实id，而是存储的使用 `LocalIndexEncoder` 生成的编码，这样节省了空间，格式为 `UncompressedInBlock` : `（商品id集，用户id集对应的编码集，打分集）` ，如， `([v1, v2, v1, v2, v2], [ui1, ui1, ui2, ui2, ui3], [r11, r12, r21, r22, r32])` 。这种结构仍旧有压缩的空间， `spark` 调用 `compress` 方法将商品 `id` 进行排序（排序有两个好处，除了压缩以外，后文构建最小二乘也会因此受益）， 并且转换为 `（不重复的有序的商品id集，商品位置偏移集，用户id集对应的编码集，打分集）` 的形式，以获得更优的存储效率（代码中就是将矩阵的 `coo` 格式转换为 `csc` 格式，你可以更进一步了解矩阵存储，以获得更多信息）。 以这样的格式修改 `([v1, v2, v1, v2, v2], [ui1, ui1, ui2, ui2, ui3], [r11, r12, r21, r22, r32])` ，得到的结果是 `([v1, v2], [0, 2, 5], [ui1, ui2, ui1, ui2, ui3], [r11, r21, r12, r22, r32])` 。其中 `[0, 2]` 指 `v1` 对应的打分的区间是 `[0, 2]` ， `[2, 5]` 指 `v2` 对应的打分的区间是 `[2, 5]` 。

`Compress` 方法利用 `spark` 内置的 `Timsort` 算法将 `UncompressedInBlock` 进行排序并转换为 `InBlock` 。代码如下所示：

```
def compress(): InBlock[ID] = {
  val sz = length
  //Timsort排序
  sort()
  val uniqueSrcIdsBuilder = mutable.ArrayBuilder.make[ID]
```

```
    val dstCountsBuilder = mutable.ArrayBuilder.make[Int]
    var preSrcId = srcIds(0)
    uniqueSrcIdsBuilder += preSrcId
    var curCount = 1
    var i = 1
    var j = 0
    while (i < sz) {
      val srcId = srcIds(i)
      if (srcId != preSrcId) {
        uniqueSrcIdsBuilder += srcId
        dstCountsBuilder += curCount
        preSrcId = srcId
        j += 1
        curCount = 0
      }
      curCount += 1
      i += 1
    }
    dstCountsBuilder += curCount
    val uniqueSrcIds = uniqueSrcIdsBuilder.result()
    val numUniqueSrdIds = uniqueSrcIds.length
    val dstCounts = dstCountsBuilder.result()
    val dstPtrs = new Array[Int](numUniqueSrdIds + 1)
    var sum = 0
    i = 0
    //计算偏移量
    while (i < numUniqueSrdIds) {
      sum += dstCounts(i)
      i += 1
      dstPtrs(i) = sum
    }
    InBlock(uniqueSrcIds, dstPtrs, dstEncodedIndices, ratings)
  }
  private def sort(): Unit = {
    val sz = length
    val sortId = Utils.random.nextInt()
    val sorter = new Sorter(new UncompressedInBlockSort[ID])
    sorter.sort(this, 0, length, Ordering[KeyWrapper[ID]])
    }
```

下面的代码用来求用户的 `OutBlock` 信息。

```
val outBlocks = inBlocks.mapValues { case InBlock(srcIds, dstPtr
s, dstEncodedIndices, _) =>
  val encoder = new LocalIndexEncoder(dstPart.numPartitions)
  val activeIds = Array.fill(dstPart.numPartitions)(mutable.Arra
yBuilder.make[Int])
  var i = 0
  val seen = new Array[Boolean](dstPart.numPartitions)
  while (i < srcIds.length) {
    var j = dstPtrs(i)
    ju.Arrays.fill(seen, false)
    while (j < dstPtrs(i + 1)) {
      val dstBlockId = encoder.blockId(dstEncodedIndices(j))
      if (!seen(dstBlockId)) {
        activeIds(dstBlockId) += i
        seen(dstBlockId) = true
      }
      j += 1
    }
    i += 1
  }
  activeIds.map { x =>
    x.result()
  }
}.setName(prefix + "OutBlocks")
  .persist(storageLevel)
```

这段代码中，`inBlocks` 表示用户的输入分区块，格式为 `（用户分区id，（不重复的用户id集，用户位置偏移集，商品id集对应的编码集，打分集））`。`activeIds` 表示商品分区中涉及的用户id集，也即上文所说的需要发送给确定的商品分区的用户信息。`activeIds` 是一个二维数组，第一维表示分区，第二维表示用户id集。用户 `OutBlocks` 的最终格式是 `（用户分区id，OutBlocks）`。

通过用户的 `OutBlock` 把用户信息发给商品分区，然后结合商品的 `InBlock` 信息构建最小二乘问题，我们就可以借此解得商品的极小解。反之，通过商品 `OutBlock` 把商品信息发送给用户分区，然后结合用户的 `InBlock` 信息构建最小二乘问题，我们就可以解得用户解。 第（6）步会详细介绍如何构建最小二乘。

- （**5**）初始化用户特征矩阵和商品特征矩阵。

交换最小二乘算法是分别固定用户特征矩阵和商品特征矩阵来交替计算下一次迭代的商品特征矩阵和用户特征矩阵。通过下面的代码初始化第一次迭代的特征矩阵。

```
var userFactors = initialize(userInBlocks, rank, seedGen.nextLon
g())
var itemFactors = initialize(itemInBlocks, rank, seedGen.nextLon
g())
```

初始化后的 `userFactors` 的格式是 `（用户分区id，用户特征矩阵factors）` ，其中 `factors` 是一个二维数组，第一维的长度是用户数，第二维的长度是 `rank` 数。初始化的值是异或随机数的F范式。 `itemFactors` 的初始化与此类似。

- （**6**）利用**inblock**和**outblock**信息构建最小二乘。

构建最小二乘的方法是在 `computeFactors` 方法中实现的。我们以商品 `inblock` 信息结合用户 `outblock` 信息构建最小二乘为例来说明这个过程。代码首先用用户 `outblock` 与 `userFactor` 进行 `join` 操作，然后以商品分区 `id` 为 `key` 进行分组。 每一个商品分区包含一组所需的用户分区及其对应的用户 `factor` 信息，格式即 `（用户分区id集，用户分区对应的factor集）` 。紧接着，用商品 `inblock` 信息与 `merged` 进行 `join` 操作，得到商品分区所需要的所有信息，即 `（商品inblock，（用户分区id集，用户分区对应的factor集））` 。 有了这些信息，构建最小二乘的数据就齐全了。详细代码如下：

```
val srcOut = srcOutBlocks.join(srcFactorBlocks).flatMap {
  case (srcBlockId, (srcOutBlock, srcFactors)) =>
    srcOutBlock.view.zipWithIndex.map { case (activeIndices, dst
BlockId) =>
      (dstBlockId, (srcBlockId, activeIndices.map(idx => srcFact
ors(idx))))
    }
}
val merged = srcOut.groupByKey(new ALSPartitioner(dstInBlocks.pa
rtitions.length))
dstInBlocks.join(merged)
```

我们知道求解商品值时，我们需要通过所有和商品关联的用户向量信息来构建最小二乘问题。这里有两个选择，第一是扫一遍 `InBlock` 信息，同时对所有的产品构建对应的最小二乘问题；第二是对于每一个产品，扫描 `InBlock` 信息，构建并求解其对应的最小二乘问题。第一种方式复杂度较高，具体的复杂度计算在此不作推导。`spark` 选取第二种方法求解最小二乘问题，同时也做了一些优化。做优化的原因是二种方法针对每个商品，都会扫描一遍 `InBlock` 信息，这会浪费较多时间，为此，将 `InBlock` 按照商品 `id` 进行排序（前文已经提到过），我们通过一次扫描就可以创建所有的最小二乘问题并求解。构建代码如下所示：

```
while (j < dstIds.length) {
  ls.reset()
  var i = srcPtrs(j)
  var numExplicits = 0
  while (i < srcPtrs(j + 1)) {
    val encoded = srcEncodedIndices(i)
    val blockId = srcEncoder.blockId(encoded)
    val localIndex = srcEncoder.localIndex(encoded)
    val srcFactor = sortedSrcFactors(blockId)(localIndex)
    val rating = ratings(i)
    ls.add(srcFactor, rating)
    numExplicits += 1
    i += 1
  }
   dstFactors(j) = solver.solve(ls, numExplicits * regParam)
  j += 1
}
```

到了这一步，构建显式反馈算法的最小二乘就结束了。隐式反馈算法的实现与此类似，不同的地方是它将 `YtY` 这个值预先计算了（可以参考文献【1】了解更多信息），而不用在每次迭代中都计算一遍。代码如下：

```
//在循环之外计算
val YtY = if (implicitPrefs) Some(computeYtY(srcFactorBlocks, ra
nk)) else None

//在每个循环内
if (implicitPrefs) {
  ls.merge(YtY.get)
}
if (implicitPrefs) {
  // Extension to the original paper to handle b < 0. confidence
 is a function of |b|
  // instead so that it is never negative. c1 is confidence - 1.
0.
  val c1 = alpha * math.abs(rating)
  // For rating <= 0, the corresponding preference is 0. So the
term below is only added
  // for rating > 0. Because YtY is already added, we need to ad
just the scaling here.
  if (rating > 0) {
    numExplicits += 1
    ls.add(srcFactor, (c1 + 1.0) / c1, c1)
  }
}
```

后面的问题就如何求解最小二乘了。我们会在最优化章节介绍`spark`版本的NNLS。

4 参考文献

【1】 Yifan Hu，Yehuda Koren∗，Chris Volinsky. Collaborative Filtering for Implicit Feedback Datasets

【2】 Yehuda Koren, Robert Bell and Chris Volinsky. Matrix Factorization Techniques for Recommender Systems

【3】 Yunhong Zhou, Dennis Wilkinson, Robert Schreiber and Rong Pan. Large-scale Parallel Collaborative Filtering for the Netflix Prize

分类与回归

`spark.mllib` 提供了多种方法用于用于二分类、多分类以及回归分析。 下表介绍了每种问题类型支持的算法。

问题类型	支持的方法
二分类	线性SVMs、逻辑回归、决策树、随机森林、梯度增强树、朴素贝叶斯
多分类	逻辑回归、决策树、随机森林、朴素贝叶斯
回归	线性最小二乘、决策树、随机森林、梯度增强树、保序回归

点击链接，了解具体的算法实现。

- 分类和回归
    - 线性模型
        - SVMs(支持向量机)
        - 逻辑回归
        - 线性回归
    - 朴素贝叶斯
    - 决策树
    - 组合树
        - 随机森林
        - 梯度提升树
    - 保序回归

线性模型

1 数学描述

许多标准的机器学习算法可以归结为凸优化问题。例如，找到凸函数 `f` 的一个极小值的任务，这个凸函数依赖于可变向量 `w` （在 `spark` 源码中，一般表示为 `weights` ）。形式上，我们可以将其当作一个凸优化问题${min}_{w}f(w)$。它的目标函数可以表示为如下公式**(1)**：

$$f(w) := \lambda R(w) + \frac{1}{n}\sum_{i=1}^{n} L(w; x_i, y_i)$$

在上式中，向量 `x` 表示训练数据集， `y` 表示它相应的标签，也是我们想预测的值。如果 `L(w;x,y)` 可以表示为${w}^{T}x$和 `y` 的函数， 我们称这个方法为线性的。 `spark.mllib` 中的几种分类算法和回归算法可以归为这一类。

目标函数 `f` 包含两部分：正则化(`regularizer`)，用于控制模型的复杂度；损失函数，用于度量模型的误差。损失函数 `L(w;.)` 是一个典型的基于 `w` 的凸函数。固定的正则化参数 `gamma` 定义了两种目标的权衡（ `trade-off` ），这两个目标分别是最小化损失(训练误差)以及最小化模型复杂度(为了避免过拟合)。

1.1 损失函数

下面介绍 `spark.mllib` 中提供的几种损失函数以及它们的梯度或子梯度(`sub-gradient`)。

- **hinge loss**

`hinge` 损失的损失函数 `L(w;x,y)` 以及梯度分别是：

$$\max\{0, 1 - yw^T x\}, y \in \{-1, +1\}$$

$$\begin{cases} -y.x, & if\ yw^T x < 1 \\ 0, & otherwise \end{cases}$$

- **logistic loss**

  `logistic` 损失的损失函数 `L(w;x,y)` 以及梯度分别是：

$$\log(1+\exp(-yw^T x)), y \in \{-1,+1\}$$

$$-y\left(1-\frac{1}{1+\exp(-yw^T x)}\right).x$$

- **squared loss**

  `squared` 损失的损失函数 `L(w;x,y)` 以及梯度分别是：

$$\frac{1}{2}(w^T x-y)^2, y \in R$$

$$(w^T x-y).x$$

1.2 正则化

正则化的目的是为了简化模型及防止过拟合。`spark.mllib` 中提供了下面的正则化方法。

问题	规则化函数**R(w)**	梯度
Zero	0	0
L2	如下公式(1)	w
L1	如下公式(2)	sign(w)
elastic net	alpha *L1* +*(1-alpha)* L2	alpha *sign(w)* + *(1-alpha)* w

$$\frac{1}{2}||w||_2^2 \qquad (1)$$

$$||w||_1 \qquad (2)$$

在上面的表格中， `sign(w)` 是一个向量，它由 `w` 中的所有实体的信号量 `(+1,-1)` 组成。 `L2` 问题往往比 `L1` 问题更容易解决，那是因为 `L2` 是平滑的。然而， `L1` 可以使权重矩阵更稀疏， 从而构建更小以及更可判断的模型，模型的可判断性在特征选择中很有用。

2 分类

分类的目的就是将数据切分为不同的类别。最一般的分类类型是二分类，即有两个类别，通常称为正和负。如果类别数超过两个，我们称之为多分类。 `spark.ml` 提供了两种线性方法用于分类：线性支持向量机以及逻辑回归。线性支持向量机仅仅支持二分类，逻辑回归既支持二分类也支持多分类。对所有的方法， `spark.ml` 支持 `L1` 和 `L2` 正则化。分类算法的详细介绍见下面的链接。

- SVMs(支持向量机)
- 逻辑回归
- 线性回归

线性支持向量机

1 介绍

线性支持向量机是一个用于大规模分类任务的标准方法。它的目标函数线性模型中的公式（1）。它的损失函数是海格损失，如下所示

$$L(\mathbf{w};\mathbf{x},y) := \max\{0, 1 - y\mathbf{w}^T\mathbf{x}\}.$$

默认情况下，线性支持向量机训练时使用 `L2` 正则化。线性支持向量机输出一个 `SVM` 模型。给定一个新的数据点 `x`，模型通过 `w^Tx` 的值预测，当这个值大于0时，输出为正，否则输出为负。

线性支持向量机并不需要核函数，要详细了解支持向量机，请参考文献【1】。

2 源码分析

2.1 实例

```
import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.util.MLUtils
// Load training data in LIBSVM format.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// Split data into training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)
// Run training algorithm to build the model
val numIterations = 100
val model = SVMWithSGD.train(training, numIterations)
// Clear the default threshold.
model.clearThreshold()
// Compute raw scores on the test set.
val scoreAndLabels = test.map { point =>
  val score = model.predict(point.features)
  (score, point.label)
}
// Get evaluation metrics.
val metrics = new BinaryClassificationMetrics(scoreAndLabels)
val auROC = metrics.areaUnderROC()
println("Area under ROC = " + auROC)
```

2.2 训练

和逻辑回归一样，训练过程均使用 `GeneralizedLinearModel` 中的 `run` 训练，只是训练使用的 `Gradient` 和 `Updater` 不同。在线性支持向量机中，使用 `HingeGradient` 计算梯度，使用 `SquaredL2Updater` 进行更新。 它的实现过程分为4步。参加逻辑回归了解这五步的详细情况。我们只需要了解 `HingeGradient` 和 `SquaredL2Updater` 的实现。

```
class HingeGradient extends Gradient {
  override def compute(data: Vector, label: Double, weights: Vec
tor): (Vector, Double) = {
    val dotProduct = dot(data, weights)
    // 我们的损失函数是 max(0, 1 - (2y - 1) (f_w(x)))
    // 所以梯度是 -(2y - 1)*x
    val labelScaled = 2 * label - 1.0
    if (1.0 > labelScaled * dotProduct) {
      val gradient = data.copy
      scal(-labelScaled, gradient)
      (gradient, 1.0 - labelScaled * dotProduct)
    } else {
      (Vectors.sparse(weights.size, Array.empty, Array.empty), 0
.0)
    }
  }

  override def compute(
      data: Vector,
      label: Double,
      weights: Vector,
      cumGradient: Vector): Double = {
    val dotProduct = dot(data, weights)
    // 我们的损失函数是 max(0, 1 - (2y - 1) (f_w(x)))
    // 所以梯度是 -(2y - 1)*x
    val labelScaled = 2 * label - 1.0
    if (1.0 > labelScaled * dotProduct) {
      //cumGradient -= labelScaled * data
      axpy(-labelScaled, data, cumGradient)
      //损失值
      1.0 - labelScaled * dotProduct
    } else {
      0.0
    }
  }
}
```

线性支持向量机的训练使用 `L2` 正则化方法。

```
class SquaredL2Updater extends Updater {
  override def compute(
      weightsOld: Vector,
      gradient: Vector,
      stepSize: Double,
      iter: Int,
      regParam: Double): (Vector, Double) = {
    // w' = w - thisIterStepSize * (gradient + regParam * w)
    // w' = (1 - thisIterStepSize * regParam) * w - thisIterStep
Size * gradient
    //表示步长，即负梯度方向的大小
    val thisIterStepSize = stepSize / math.sqrt(iter)
    val brzWeights: BV[Double] = weightsOld.toBreeze.toDenseVect
or
    //正则化，brzWeights每行数据均乘以(1.0 - thisIterStepSize * regP
aram)
    brzWeights :*= (1.0 - thisIterStepSize * regParam)
    //y += x * a，即brzWeights -= gradient * thisInterStepSize
    brzAxpy(-thisIterStepSize, gradient.toBreeze, brzWeights)
    //正则化||w||_2
    val norm = brzNorm(brzWeights, 2.0)
    (Vectors.fromBreeze(brzWeights), 0.5 * regParam * norm * nor
m)
  }
}
```

该函数的实现规则是：

```
w' = w - thisIterStepSize * (gradient + regParam * w)
w' = (1 - thisIterStepSize * regParam) * w - thisIterStepSize *
gradient
```

这里 `thisIterStepSize` 表示参数沿负梯度方向改变的速率，它随着迭代次数的增多而减小。

2.3 预测

```
override protected def predictPoint(
      dataMatrix: Vector,
      weightMatrix: Vector,
      intercept: Double) = {
    //w^Tx
    val margin = weightMatrix.toBreeze.dot(dataMatrix.toBreeze)
+ intercept
    threshold match {
      case Some(t) => if (margin > t) 1.0 else 0.0
      case None => margin
    }
  }
```

参考文献

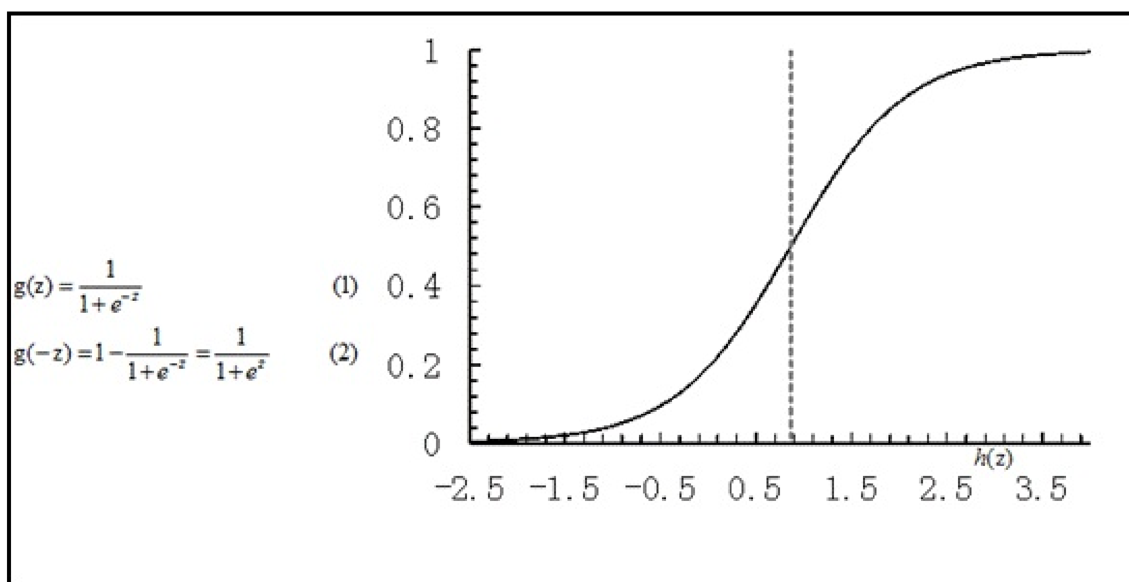
【1】支持向量机通俗导论（理解SVM的三层境界）

逻辑回归

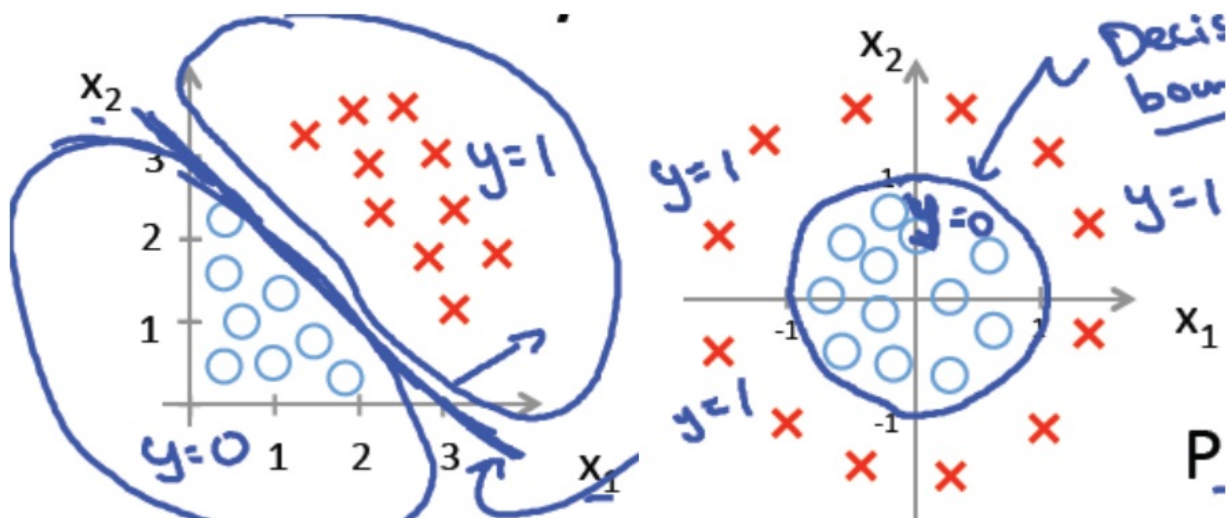
1 二元逻辑回归

回归是一种很容易理解的模型，就相当于 `y=f(x)` ，表明自变量 `x` 与因变量 `y` 的关系。最常见问题如医生治病时的望、闻、问、切，之后判定病人是否生病或生了什么病， 其中的望、闻、问、切就是获取的自变量 `x` ，即特征数据，判断是否生病就相当于获取因变量 `y` ，即预测分类。最简单的回归是线性回归，但是线性回归的鲁棒性很差。

逻辑回归是一种减小预测范围，将预测值限定为 `[0,1]` 间的一种回归模型，其回归方程与回归曲线如下图所示。逻辑曲线在 `z=0` 时，十分敏感，在 `z>>0` 或 `z<<0` 时，都不敏感。



逻辑回归其实是在线性回归的基础上，套用了一个逻辑函数。上图的 `g(z)` 就是这个逻辑函数(或称为 `Sigmoid` 函数)。下面左图是一个线性的决策边界，右图是非线性的决策边界。



对于线性边界的情况，边界形式可以归纳为如下公式**(1)**:

$$\theta_0+\theta_1 x_1+,\ldots,+\theta_n x_n=\sum_{i=1}^{n}\theta_i x_i=\theta^T x$$

因此我们可以构造预测函数为如下公式**(2)**:

$$h_\theta(x)=g(\theta^T x)=\frac{1}{1+e^{-\theta^T x}}$$

该预测函数表示分类结果为1时的概率。因此对于输入点 `x` ，分类结果为类别1和类别0的概率分别为如下公式**(3)**：

$$P(y=1|x;\theta)=h_\theta(x)$$
$$P(y=0|x;\theta)=1-h_\theta(x)$$

对于训练数据集，特征数据 `x={x1, x2, … , xm}` 和对应的分类数据 `y={y1, y2, … , ym}` 。构建逻辑回归模型 `f` ，最典型的构建方法便是应用极大似然估计。对公式**(3)**取极大似然函数，可以得到如下的公式**(4)**:

$$L(\theta)=\prod_{i=1}^{m}P(y_i|x_i;\theta)=\prod_{i=1}^{m}(h_\theta(x_i))^{y_i}(1-h_\theta(x_i))^{1-y_i}$$

再对公式**(4)**取对数，可得到公式**(5)**：

$$l(\theta)=\log L(\theta)=\sum_{i=1}^{m}\left(y_i\log h_\theta(x_i)+(1-y_i)\log(1-h_\theta(x_i))\right)$$

最大似然估计就是求使 `l` 取最大值时的 `theta` 。 `MLlib` 中提供了两种方法来求这个参数，分别是梯度下降法和L-BFGS。

2 多元逻辑回归

二元逻辑回归可以一般化为多元逻辑回归用来训练和预测多分类问题。对于多分类问题，算法将会训练出一个多元逻辑回归模型， 它包含 `K-1` 个二元回归模型。给定一个数据点， `K-1` 个模型都会运行，概率最大的类别将会被选为预测类别。

对于输入点 `x` ，分类结果为各类别的概率分别为如下公式**(6)**，其中 `k` 表示类别个数。

$$\mathrm{p}(y=0|x;w)=\frac{1}{1+\sum_{i=1}^{k-1}e^{xw_i}}$$

$$\mathrm{p}(y=1|x;w)=\frac{e^{xw_1}}{1+\sum_{i=1}^{k-1}e^{xw_i}}$$

$$\mathrm{p}(y=k-1|x;w)=\frac{e^{xw_{k-1}}}{1+\sum_{i=1}^{k-1}e^{xw_i}}$$

对于 `k` 类的多分类问题，模型的权重 `w = (w_1, w_2, ..., w_{K-1})` 是一个矩阵，如果添加截距，矩阵的维度为 `(K-1) * (N+1)` ，否则为 `(K-1) * N` 。单个样本的目标函数的损失函数可以写成如下公式**(7)**的形式。

$$\mathrm{l}(w,x)=-\mathrm{logp}(y|x,w)=-\alpha_y logp(y=0|x;w)-(1-\alpha_y)logp(y|x;w)$$

$$=\log\left(1+\sum_{i=1}^{k-1}e^{xw_i}\right)-(1-\alpha_y)xw_{y-1}=\log\left(1+\sum_{i=1}^{k-1}e^{margin\,s_i}\right)-(1-\alpha_y)margin\,s_{y-1}$$

$$,\begin{cases}\alpha_y=1, if\ y=0\\ \alpha_y=0, if\ y\neq 0\end{cases}$$

对损失函数求一阶导数，我们可以得到下面的公式**(8)**:

$$\frac{\rho l(w,x)}{\rho w_{ij}}=\left(\frac{e^{margin\, s_i}}{1+\sum_{i=1}^{k-1}e^{margin\, s_i}}-(1-\alpha_y)\delta_{y,i+1}\right)x_j=multifier_i * x_j$$

$$\delta_{i,j}=1, if\ i=j$$
$$\delta_{i,j}=0, if\ i\neq j$$

根据上面的公式，如果某些`margin`的值大于709.78，`multiplier`以及逻辑函数的计算会出现算术溢出(`arithmetic overflow`)的情况。这个问题发生在有离群点远离超平面的情况下。幸运的是，当`max(margins) = maxMargin > 0`时，损失函数可以重写为如下公式**(9)**的形式。

$$\mathrm{l(w,x)}=\log\left(1+\sum_{i=1}^{k-1}e^{margin\, s_i}\right)-(1-\alpha_y)margin\, s_{y-1}$$

$$=\log(\mathrm{e}^{-maxMargin}+\sum_{i=1}^{k-1}e^{margin\, s_i-maxMargin})+\mathrm{maxMargin}-(1-\alpha_y)margin\, s_{y-1}$$

$$=\log(1+\mathrm{sum})+\mathrm{maxMargin}-(1-\alpha_y)margin\, s_{y-1}$$

同理，`multiplier`也可以重写为如下公式**(10)**的形式。

$$multifier=\frac{e^{margin\, s_i}}{1+\sum_{i=1}^{k-1}e^{margin\, s_i}}-(1-\alpha_y)\delta_{y,i+1}$$

$$=\frac{e^{(margin\, s_i-maxMargin)}}{\mathrm{e}^{-maxMargin}+\sum_{i=1}^{k-1}e^{margin\, s_i-maxMargin}}-(1-\alpha_y)\delta_{y,i+1}$$

$$=\frac{e^{(margin\, s_i-maxMargin)}}{1+\mathrm{sum}}-(1-\alpha_y)\delta_{y,i+1}$$

3 逻辑回归的优缺点

- 优点：计算代价低，速度快，容易理解和实现。
- 缺点：容易欠拟合，分类和回归的精度不高。

4 实例

下面的例子展示了如何使用逻辑回归。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.classification.{LogisticRegression
WithLBFGS, LogisticRegressionModel}
import org.apache.spark.mllib.evaluation.MulticlassMetrics
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.util.MLUtils
// 加载训练数据
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_
data.txt")
// 切分数据，training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)
// 训练模型
val model = new LogisticRegressionWithLBFGS()
  .setNumClasses(10)
  .run(training)
// Compute raw scores on the test set.
val predictionAndLabels = test.map { case LabeledPoint(label, fe
atures) =>
  val prediction = model.predict(features)
  (prediction, label)
}
// Get evaluation metrics.
val metrics = new MulticlassMetrics(predictionAndLabels)
val precision = metrics.precision
println("Precision = " + precision)
// 保存和加载模型
model.save(sc, "myModelPath")
val sameModel = LogisticRegressionModel.load(sc, "myModelPath")
```

5 源码分析

5.1 训练模型

如上所述，在 `MLlib` 中，分别使用了梯度下降法和 `L-BFGS` 实现逻辑回归参数的计算。这两个算法的实现我们会在最优化章节介绍，这里我们介绍公共的部分。

`LogisticRegressionWithLBFGS` 和 `LogisticRegressionWithSGD` 的入口函数均是 `GeneralizedLinearAlgorithm.run` ，下面详细分析该方法。

```
def run(input: RDD[LabeledPoint]): M = {
    if (numFeatures < 0) {
      //计算特征数
      numFeatures = input.map(_.features.size).first()
    }
    val initialWeights = {
          if (numOfLinearPredictor == 1) {
            Vectors.zeros(numFeatures)
          } else if (addIntercept) {
            Vectors.zeros((numFeatures + 1) * numOfLinearPredict
or)
          } else {
            Vectors.zeros(numFeatures * numOfLinearPredictor)
          }
    }
    run(input, initialWeights)
}
```

上面的代码初始化权重向量，向量的值均初始化为0。需要注意的是， `addIntercept` 表示是否添加截距(`Intercept` ，指函数图形与坐标的交点到原点的距离)，默认是不添加的。 `numOfLinearPredictor` 表示二元逻辑回归模型的个数。 我们重点看 `run(input, initialWeights)` 的实现。它的实现分四步。

5.1.1 根据提供的参数缩放特征并添加截距

```
val scaler = if (useFeatureScaling) {
      new StandardScaler(withStd = true, withMean = false).fit(input.map(_.features))
    } else {
      null
    }
val data =
      if (addIntercept) {
        if (useFeatureScaling) {
          input.map(lp => (lp.label, appendBias(scaler.transform(lp.features)))).cache()
        } else {
          input.map(lp => (lp.label, appendBias(lp.features))).cache()
        }
      } else {
        if (useFeatureScaling) {
          input.map(lp => (lp.label, scaler.transform(lp.features))).cache()
        } else {
          input.map(lp => (lp.label, lp.features))
        }
      }
val initialWeightsWithIntercept = if (addIntercept && numOfLinearPredictor == 1) {
      appendBias(initialWeights)
    } else {
      /** If `numOfLinearPredictor > 1`, initialWeights already contains intercepts. */
      initialWeights
    }
```

在最优化过程中，收敛速度依赖于训练数据集的条件数(`condition number`)，缩放变量经常可以启发式地减少这些条件数，提高收敛速度。不减少条件数，一些混合有不同范围列的数据集可能不能收敛。 在这里使用 `StandardScaler` 将数据集的特征进行缩放。详细信息请看StandardScaler。 `appendBias` 方法很简单，就是在每个向量后面加一个值为1的项。

```
def appendBias(vector: Vector): Vector = {
    vector match {
      case dv: DenseVector =>
        val inputValues = dv.values
        val inputLength = inputValues.length
        val outputValues = Array.ofDim[Double](inputLength + 1)
        System.arraycopy(inputValues, 0, outputValues, 0, inputL
ength)
        outputValues(inputLength) = 1.0
        Vectors.dense(outputValues)
      case sv: SparseVector =>
        val inputValues = sv.values
        val inputIndices = sv.indices
        val inputValuesLength = inputValues.length
        val dim = sv.size
        val outputValues = Array.ofDim[Double](inputValuesLength
 + 1)
        val outputIndices = Array.ofDim[Int](inputValuesLength +
1)
        System.arraycopy(inputValues, 0, outputValues, 0, inputV
aluesLength)
        System.arraycopy(inputIndices, 0, outputIndices, 0, inpu
tValuesLength)
        outputValues(inputValuesLength) = 1.0
        outputIndices(inputValuesLength) = dim
        Vectors.sparse(dim + 1, outputIndices, outputValues)
      case _ => throw new IllegalArgumentException(s"Do not supp
ort vector type ${vector.getClass}")
    }
```

5.1.2 使用最优化算法计算最终的权重值

```
val weightsWithIntercept = optimizer.optimize(data, initialWeigh
tsWithIntercept)
```

有梯度下降算法和 `L-BFGS` 两种算法来计算最终的权重值，查看梯度下降法和L-BFGS了解详细实现。 这两种算法均使用 `Gradient` 的实现类计算梯度，使用 `Updater` 的实现类更新参数。在 `LogisticRegressionWithSGD` 和 `LogisticRegressionWithLBFGS` 中，它们均使用 `LogisticGradient` 实现类计算梯度，使用 `SquaredL2Updater` 实现类更新参数。

```
//在GradientDescent中
private val gradient = new LogisticGradient()
private val updater = new SquaredL2Updater()
override val optimizer = new GradientDescent(gradient, updater)
    .setStepSize(stepSize)
    .setNumIterations(numIterations)
    .setRegParam(regParam)
    .setMiniBatchFraction(miniBatchFraction)
//在LBFGS中
override val optimizer = new LBFGS(new LogisticGradient, new Squ
aredL2Updater)
```

下面将详细介绍 `LogisticGradient` 的实现和 `SquaredL2Updater` 的实现。

- LogisticGradient

`LogisticGradient` 中使用 `compute` 方法计算梯度。计算分为两种情况，即二元逻辑回归的情况和多元逻辑回归的情况。虽然多元逻辑回归也可以实现二元分类，但是为了效率， `compute` 方法仍然实现了一个二元逻辑回归的版本。

```
val margin = -1.0 * dot(data, weights)
val multiplier = (1.0 / (1.0 + math.exp(margin))) - label
//y += a * x，即cumGradient += multiplier * data
axpy(multiplier, data, cumGradient)
if (label > 0) {
    // The following is equivalent to log(1 + exp(margin)) but m
ore numerically stable.
    MLUtils.log1pExp(margin)
} else {
    MLUtils.log1pExp(margin) - margin
}
```

这里的 `multiplier` 就是上文的公式**(2)**。 `axpy` 方法用于计算梯度，这里表示的意思是 `h(x) * x` 。下面是多元逻辑回归的实现方法。

```
//权重
val weightsArray = weights match {
    case dv: DenseVector => dv.values
    case _ =>
            throw new IllegalArgumentException
}
//梯度
val cumGradientArray = cumGradient match {
    case dv: DenseVector => dv.values
    case _ =>
        throw new IllegalArgumentException
}
// 计算所有类别中最大的margin
var marginY = 0.0
var maxMargin = Double.NegativeInfinity
var maxMarginIndex = 0
val margins = Array.tabulate(numClasses - 1) { i =>
    var margin = 0.0
    data.foreachActive { (index, value) =>
        if (value != 0.0) margin += value * weightsArray((i * da
taSize) + index)
    }
    if (i == label.toInt - 1) marginY = margin
    if (margin > maxMargin) {
            maxMargin = margin
            maxMarginIndex = i
    }
    margin
}
//计算sum，保证每个margin都小于0，避免出现算术溢出的情况
val sum = {
     var temp = 0.0
     if (maxMargin > 0) {
         for (i <- 0 until numClasses - 1) {
              margins(i) -= maxMargin
              if (i == maxMarginIndex) {
                temp += math.exp(-maxMargin)
```

```
              } else {
                temp += math.exp(margins(i))
              }
          }
      } else {
          for (i <- 0 until numClasses - 1) {
              temp += math.exp(margins(i))
          }
      }
      temp
}
//计算multiplier并计算梯度
for (i <- 0 until numClasses - 1) {
      val multiplier = math.exp(margins(i)) / (sum + 1.0) - {
           if (label != 0.0 && label == i + 1) 1.0 else 0.0
      }
      data.foreachActive { (index, value) =>
          if (value != 0.0) cumGradientArray(i * dataSize + index
) += multiplier * value
      }
}
//计算损失函数,
val loss = if (label > 0.0) math.log1p(sum) - marginY else math.
log1p(sum)
if (maxMargin > 0) {
      loss + maxMargin
} else {
      loss
}
```

- SquaredL2Updater

```
class SquaredL2Updater extends Updater {
  override def compute(
      weightsOld: Vector,
      gradient: Vector,
      stepSize: Double,
      iter: Int,
      regParam: Double): (Vector, Double) = {
    // w' = w - thisIterStepSize * (gradient + regParam * w)
    // w' = (1 - thisIterStepSize * regParam) * w - thisIterStepSize * gradient
    //表示步长，即负梯度方向的大小
    val thisIterStepSize = stepSize / math.sqrt(iter)
    val brzWeights: BV[Double] = weightsOld.toBreeze.toDenseVector
    //正则化，brzWeights每行数据均乘以(1.0 - thisIterStepSize * regParam)
    brzWeights :*= (1.0 - thisIterStepSize * regParam)
    //y += x * a，即brzWeights -= gradient * thisInterStepSize
    brzAxpy(-thisIterStepSize, gradient.toBreeze, brzWeights)
    //正则化||w||_2
    val norm = brzNorm(brzWeights, 2.0)
    (Vectors.fromBreeze(brzWeights), 0.5 * regParam * norm * norm)
  }
}
```

该函数的实现规则是：

```
w' = w - thisIterStepSize * (gradient + regParam * w)
w' = (1 - thisIterStepSize * regParam) * w - thisIterStepSize * gradient
```

这里 `thisIterStepSize` 表示参数沿负梯度方向改变的速率，它随着迭代次数的增多而减小。

5.1.3 对最终的权重值进行后处理

```
val intercept = if (addIntercept && numOfLinearPredictor == 1) {
      weightsWithIntercept(weightsWithIntercept.size - 1)
    } else {
      0.0
    }
var weights = if (addIntercept && numOfLinearPredictor == 1) {
      Vectors.dense(weightsWithIntercept.toArray.slice(0, weight
sWithIntercept.size - 1))
    } else {
      weightsWithIntercept
    }
```

该段代码获得了截距（`intercept`）以及最终的权重值。由于截距（`intercept`）和权重是在收缩的空间进行训练的，所以我们需要再把它们转换到原始的空间。数学知识告诉我们，如果我们仅仅执行标准化而没有减去均值，即`withStd = true, withMean = false`，那么截距（`intercept`）的值并不会发送改变。所以下面的代码仅仅处理权重向量。

```
if (useFeatureScaling) {
      if (numOfLinearPredictor == 1) {
        weights = scaler.transform(weights)
      } else {
        var i = 0
        val n = weights.size / numOfLinearPredictor
        val weightsArray = weights.toArray
        while (i < numOfLinearPredictor) {
          //排除intercept
          val start = i * n
          val end = (i + 1) * n - { if (addIntercept) 1 else 0 }
          val partialWeightsArray = scaler.transform(
            Vectors.dense(weightsArray.slice(start, end))).toArr
ay
          System.arraycopy(partialWeightsArray, 0, weightsArray,
 start, partialWeightsArray.size)
          i += 1
        }
        weights = Vectors.dense(weightsArray)
      }
    }
```

5.1.4 创建模型

```
createModel(weights, intercept)
```

5.2 预测

训练完模型之后，我们就可以通过训练的模型计算得到测试数据的分类信息。 `predictPoint` 用来预测分类信息。它针对二分类和多分类，分别进行处理。

- 二分类的情况

```
val margin = dot(weightMatrix, dataMatrix) + intercept
val score = 1.0 / (1.0 + math.exp(-margin))
threshold match {
    case Some(t) => if (score > t) 1.0 else 0.0
    case None => score
}
```

我们可以看到 `1.0 / (1.0 + math.exp(-margin))` 就是上文提到的逻辑函数即 `sigmoid` 函数。

- 多分类情况

```
var bestClass = 0
var maxMargin = 0.0
val withBias = dataMatrix.size + 1 == dataWithBiasSize
(0 until numClasses - 1).foreach { i =>
        var margin = 0.0
        dataMatrix.foreachActive { (index, value) =>
          if (value != 0.0) margin += value * weightsArray((i *
dataWithBiasSize) + index)
        }
        // Intercept is required to be added into margin.
        if (withBias) {
          margin += weightsArray((i * dataWithBiasSize) + dataMa
trix.size)
        }
        if (margin > maxMargin) {
          maxMargin = margin
          bestClass = i + 1
        }
}
bestClass.toDouble
```

该段代码计算并找到最大的 `margin` 。如果 `maxMargin` 为负，那么第一类是该数据的类别。

参考文献

【1】逻辑回归模型(Logistic Regression, LR)基础

【2】逻辑回归

线性回归

回归问题的条件或者说前提是

- 1） 收集的数据
- 2） 假设的模型，即一个函数，这个函数里含有未知的参数，通过学习，可以估计出参数。然后利用这个模型去预测/分类新的数据。

1 线性回归的概念

线性回归假设特征和结果都满足线性。即不大于一次方。收集的数据中，每一个分量，就可以看做一个特征数据。每个特征至少对应一个未知的参数。这样就形成了一个线性模型函数，向量表示形式：

$$h_\theta(x) = \theta^T X$$

这个就是一个组合问题，已知一些数据，如何求里面的未知参数，给出一个最优解。 一个线性矩阵方程，直接求解，很可能无法直接求解。有唯一解的数据集，微乎其微。

基本上都是解不存在的超定方程组。因此，需要退一步，将参数求解问题，转化为求最小误差问题，求出一个最接近的解，这就是一个松弛求解。

在回归问题中，线性最小二乘是最普遍的求最小误差的形式。它的损失函数就是二乘损失。如下公式（**1**）所示：

$$L(\mathbf{w}; \mathbf{x}, y) := \frac{1}{2}(\mathbf{w}^T\mathbf{x} - y)^2.$$

根据使用的正则化类型的不同，回归算法也会有不同。普通最小二乘和线性最小二乘回归不使用正则化方法。`ridge` 回归使用 `L2` 正则化，`lasso` 回归使用 `L1` 正则化。

2 线性回归源码分析

2.1 实例

```
import org.apache.spark.ml.regression.LinearRegression

// 加载数据
val training = spark.read.format("libsvm")
  .load("data/mllib/sample_linear_regression_data.txt")

val lr = new LinearRegression()
  .setMaxIter(10)
  .setRegParam(0.3)
  .setElasticNetParam(0.8)

// 训练模型
val lrModel = lr.fit(training)

// 打印线性回归的系数和截距
println(s"Coefficients: ${lrModel.coefficients} Intercept: ${lrM
odel.intercept}")

// 打印统计信息
val trainingSummary = lrModel.summary
println(s"numIterations: ${trainingSummary.totalIterations}")
println(s"objectiveHistory: [${trainingSummary.objectiveHistory.
mkString(",")}]")
trainingSummary.residuals.show()
println(s"RMSE: ${trainingSummary.rootMeanSquaredError}")
println(s"r2: ${trainingSummary.r2}")
```

2.2 代码实现

2.2.1 参数配置

根据上面的例子，我们先看看线性回归可以配置的参数。

```
// 正则化参数，默认为0，对应于优化算法中的lambda
def setRegParam(value: Double): this.type = set(regParam, value)
setDefault(regParam -> 0.0)

// 是否使用截距，默认使用
```

```
def setFitIntercept(value: Boolean): this.type = set(fitIntercep
t, value)
setDefault(fitIntercept -> true)

// 在训练模型前，是否对训练特征进行标准化。默认使用。
// 模型的相关系数总是会返回回原来的空间（不是标准化后的标准空间），所以这个过
程对用户透明
def setStandardization(value: Boolean): this.type = set(standard
ization, value)
setDefault(standardization -> true)

// ElasticNet混合参数
// 当改值为0时，使用L2惩罚；当该值为1时，使用L1惩罚；当值在(0,1)之间时，使
用L1惩罚和L2惩罚的组合
def setElasticNetParam(value: Double): this.type = set(elasticNe
tParam, value)
setDefault(elasticNetParam -> 0.0)

// 最大迭代次数，默认是100
def setMaxIter(value: Int): this.type = set(maxIter, value)
setDefault(maxIter -> 100)

// 收敛阈值
def setTol(value: Double): this.type = set(tol, value)
setDefault(tol -> 1E-6)

// 样本权重列的列名。默认不设置。当不设置时，样本权重为1
def setWeightCol(value: String): this.type = set(weightCol, valu
e)

// 最优化求解方法。实际有l-bfgs和带权最小二乘两种求解方法。
// 当特征列数量超过4096时，默认使用l-bfgs求解，否则使用带权最小二乘求解。
def setSolver(value: String): this.type = {
    require(Set("auto", "l-bfgs", "normal").contains(value),
      s"Solver $value was not supported. Supported options: auto
, l-bfgs, normal")
    set(solver, value)
  }
setDefault(solver -> "auto")
```

```
// 设置treeAggregate的深度。默认情况下深度为2
// 当特征维度较大或者分区较多时，可以调大该深度
def setAggregationDepth(value: Int): this.type = set(aggregation
Depth, value)
setDefault(aggregationDepth -> 2)
```

2.2.2 训练模型

`train` 方法训练模型并返回 `LinearRegressionModel` 。方法的开始是处理数据集，生成需要的 `RDD` 。

```
// Extract the number of features before deciding optimization s
olver.
val numFeatures = dataset.select(col($(featuresCol))).first().ge
tAs[Vector](0).size
val w = if (!isDefined(weightCol) || $(weightCol).isEmpty) lit(1
.0) else col($(weightCol))

val instances: RDD[Instance] = dataset.select(
    col($(labelCol)), w, col($(featuresCol))).rdd.map {
    case Row(label: Double, weight: Double, features: Vector) =>
        Instance(label, weight, features)  // 标签，权重，特征向量
}
```

2.2.2.1 带权最小二乘

当样本的特征维度小于4096并且 `solver` 为 `auto` 或者 `solver` 为 `normal` 时，用 `WeightedLeastSquares` 求解，这是因为 `WeightedLeastSquares` 只需要处理一次数据， 求解效率更高。 `WeightedLeastSquares` 的介绍见带权最小二乘。

```
if (($(solver) == "auto" &&
    numFeatures <= WeightedLeastSquares.MAX_NUM_FEATURES) || $(s
olver) == "normal") {

    val optimizer = new WeightedLeastSquares($(fitIntercept), $(
regParam),
        elasticNetParam = $(elasticNetParam), $(standardization)
, true,
        solverType = WeightedLeastSquares.Auto, maxIter = $(maxI
ter), tol = $(tol))
    val model = optimizer.fit(instances)
    // When it is trained by WeightedLeastSquares, training summ
ary does not
    // attach returned model.
    val lrModel = copyValues(new LinearRegressionModel(uid, mode
l.coefficients, model.intercept))
    val (summaryModel, predictionColName) = lrModel.findSummaryM
odelAndPredictionCol()
    val trainingSummary = new LinearRegressionTrainingSummary(
        summaryModel.transform(dataset),
        predictionColName,
        $(labelCol),
        $(featuresCol),
        summaryModel,
        model.diagInvAtWA.toArray,
        model.objectiveHistory)

    return lrModel.setSummary(Some(trainingSummary))
}
```

2.2.2.2 拟牛顿法

- **1** 统计样本指标

当样本的特征维度大于4096并且 `solver` 为 `auto` 或者 `solver` 为 `l-bfgs` 时，使用拟牛顿法求解最优解。使用拟牛顿法求解之前我们 需要先统计特征和标签的相关信息。

```
val (featuresSummarizer, ySummarizer) = {
      val seqOp = (c: (MultivariateOnlineSummarizer, Multivariat
eOnlineSummarizer),
        instance: Instance) =>
          (c._1.add(instance.features, instance.weight),
            c._2.add(Vectors.dense(instance.label), instance.wei
ght))

      val combOp = (c1: (MultivariateOnlineSummarizer, Multivari
ateOnlineSummarizer),
        c2: (MultivariateOnlineSummarizer, MultivariateOnlineSum
marizer)) =>
          (c1._1.merge(c2._1), c1._2.merge(c2._2))

      instances.treeAggregate(
        new MultivariateOnlineSummarizer, new MultivariateOnline
Summarizer
      )(seqOp, combOp, $(aggregationDepth))
}
```

这里 `MultivariateOnlineSummarizer` 继承自 `MultivariateStatisticalSummary` ，它使用在线（ `online` ）的方式统计样本的均值、方差、最小值、最大值等指标。 具体的实现见 `MultivariateOnlineSummarizer` 。统计好指标之后，根据指标的不同选择不同的处理方式。

如果标签的方差为0，并且不管我们是否选择使用偏置，系数均为0，此时并不需要训练模型。

```
 val coefficients = Vectors.sparse(numFeatures, Seq())  // 系数为空

 val intercept = yMean
 val model = copyValues(new LinearRegressionModel(uid, coefficie
nts, intercept))
```

获取标签方差，特征均值、特征方差以及正则化项。

```
 // if y is constant (rawYStd is zero), then y cannot be scaled.
 In this case
 // setting yStd=abs(yMean) ensures that y is not scaled anymore
 in l-bfgs algorithm.
 val yStd = if (rawYStd > 0) rawYStd else math.abs(yMean)
 val featuresMean = featuresSummarizer.mean.toArray
 val featuresStd = featuresSummarizer.variance.toArray.map(math.
sqrt)
 val bcFeaturesMean = instances.context.broadcast(featuresMean)
 val bcFeaturesStd = instances.context.broadcast(featuresStd)

 val effectiveRegParam = $(regParam) / yStd
 val effectiveL1RegParam = $(elasticNetParam) * effectiveRegPara
m
 val effectiveL2RegParam = (1.0 - $(elasticNetParam)) * effectiv
eRegParam
```

- **2** 定义损失函数

```
val costFun = new LeastSquaresCostFun(instances, yStd, yMean, $(
fitIntercept),
      $(standardization), bcFeaturesStd, bcFeaturesMean, effecti
veL2RegParam, $(aggregationDepth))
```

损失函数 `LeastSquaresCostFun` 继承自 `DiffFunction[T]` ，用于表示最小二乘损失。它返回一个点L2正则化后的损失和梯度。 它使用方法 `def calculate(coefficients: BDV[Double]): (Double, BDV[Double])` 计算损失和梯度。这里 `coefficients` 表示一个特定的点。

```
override def calculate(coefficients: BDV[Double]): (Double, BDV[
Double]) = {
    val coeffs = Vectors.fromBreeze(coefficients)
    val bcCoeffs = instances.context.broadcast(coeffs)
    val localFeaturesStd = bcFeaturesStd.value

    val leastSquaresAggregator = {
      val seqOp = (c: LeastSquaresAggregator, instance: Instance
) => c.add(instance)
```

```
      val combOp = (c1: LeastSquaresAggregator, c2: LeastSquares
Aggregator) => c1.merge(c2)

      instances.treeAggregate(
        new LeastSquaresAggregator(bcCoeffs, labelStd, labelMean
, fitIntercept, bcFeaturesStd,
          bcFeaturesMean))(seqOp, combOp, aggregationDepth)
    }

    val totalGradientArray = leastSquaresAggregator.gradient.toA
rray //梯度
    bcCoeffs.destroy(blocking = false)

    val regVal = if (effectiveL2regParam == 0.0) {
      0.0
    } else {
      var sum = 0.0
      coeffs.foreachActive { (index, value) =>
        // 下面的代码计算正则化项的损失和梯度，并将梯度添加到totalGradie
ntArray中
        sum += {
          if (standardization) {
            totalGradientArray(index) += effectiveL2regParam * v
alue
            value * value
          } else {
            if (localFeaturesStd(index) != 0.0) {
              // 如果`standardization`为false，我们仍然标准化数据加快
收敛速度。获得的结果，我们需要执行反标准化
              // ，来得到正确的目标函数
              val temp = value / (localFeaturesStd(index) * loca
lFeaturesStd(index))
              totalGradientArray(index) += effectiveL2regParam *
 temp
              value * temp
            } else {
              0.0
            }
          }
        }
```

```
      }
      0.5 * effectiveL2regParam * sum
    }

    (leastSquaresAggregator.loss + regVal, new BDV(totalGradient
Array))
  }
```

这里 `LeastSquaresAggregator` 用来计算最小二乘损失函数的梯度和损失。为了在优化过程中提高收敛速度，防止大方差 的特征在训练时产生过大的影响，将特征缩放到单元方差并且减去均值，可以减少条件数。当使用截距进行训练时，处在缩放后空间的目标函数 如下：

> $$ \begin{align} L &= 1/2N ||\sum_i w_i(x_i - \bar{x_i}) / \hat{x_i} - (y - \bar{y}) / \hat{y}||^2 \end{align} $$

在这个公式中，$\bar{x_i}$是x_i的均值，$\hat{x_i}$是x_i的标准差，$\bar{y}$是标签的均值，$\hat{y}$ 是标签的标准差。

如果不使用截距，我们可以使用同样的公式。不同的是$\bar{y}$和$\bar{x_i}$分别用0代替。这个公式可以重写为如下的形式。

> $$ \begin{align} L &= 1/2N ||\sum_i (w_i/\hat{x_i})x_i - \sum_i (w_i/\hat{x_i})\bar{x_i} - y / \hat{y} + \bar{y} / \hat{y}||^2 \\ &= 1/2N ||\sum_i w_i^\prime x_i - y / \hat{y} + offset||^2 = 1/2N diff^2 \end{align} $$

在这个公式中，$w_i^\prime$是有效的相关系数，通过$w_i/\hat{x_i}$计算。 `offset` 是$- \sum_i (w_i/\hat{x_i})\bar{x_i} + \bar{y} / \hat{y}$， 而 `diff` 是$\sum_i w_i^\prime x_i - y / \hat{y} + offset$。

注意，相关系数和 `offset` 不依赖于训练数据集，所以它们可以提前计算。

现在，目标函数的一阶导数如下所示：

> $$ \begin{align} \frac{\partial L}{\partial w_i} &= diff/N (x_i - \bar{x_i}) / \hat{x_i} \end{align} $$

然而，$(x_i - \bar{x_i})$是一个密集的计算，当训练数据集是稀疏的格式时，这不是一个理想的公式。通过添加一个稠密项 $\bar{x_i} / \hat{x_i}$到 公式的末尾可以解决这个问题。目标函数的一阶导数如下所示：

> $$ \begin{align} \frac{\partial L}{\partial w_i} &=1/N \sum_j diff_j (x_{ij} - \bar{x_i}) / \hat{x_i} \\ &= 1/N ((\sum_j diff_j x_{ij} / \hat{x_i}) - diffSum \bar{x_i} / \hat{x_i}) \\ &= 1/N ((\sum_j diff_j x_{ij} / \hat{x_i}) + correction_i) \end{align} $$

这里，$correction_i = - diffSum \bar{x_i} / \hat{x_i}$。通过一个简单的数学推导，我们就可以知道 `diffSum` 实际上为0。

> $$ \begin{align} diffSum &= \sum_j (\sum_i w_i(x_{ij} - \bar{x_i}) / \hat{x_i} - (y_j - \bar{y}) / \hat{y}) \\ &= N * (\sum_i w_i(\bar{x_i} - \bar{x_i}) / \hat{x_i} - (\bar{y} - \bar{y}) / \hat{y}) \\ &= 0 \end{align} $$

所以，目标函数的一阶导数仅仅依赖于训练数据集，我们可以简单的通过分布式的方式来计算，并且对稀疏格式也很友好。

> $$ \begin{align} \frac{\partial L}{\partial w_i} &= 1/N ((\sum_j diff_j x_{ij} / \hat{x_i}) \end{align} $$

我们首先看有效系数$w_i/\hat{x_i}$和 `offset` 的实现。

```
@transient private lazy val effectiveCoefAndOffset = {
    val coefficientsArray = bcCoefficients.value.toArray.clone()
//系数，表示公式中的w
    val featuresMean = bcFeaturesMean.value
    var sum = 0.0
    var i = 0
    val len = coefficientsArray.length
    while (i < len) {
      if (featuresStd(i) != 0.0) {
        coefficientsArray(i) /=  featuresStd(i)
        sum += coefficientsArray(i) * featuresMean(i)
      } else {
        coefficientsArray(i) = 0.0
      }
      i += 1
    }
    val offset = if (fitIntercept) labelMean / labelStd - sum el
se 0.0
    (Vectors.dense(coefficientsArray), offset)
  }
```

我们再来看看 `add` 方法和 `merge` 方法的实现。当添加一个样本后，需要更新相应的损失值和梯度值。

```
def add(instance: Instance): this.type = {
  instance match { case Instance(label, weight, features) =>
  if (weight == 0.0) return this
  // 计算diff
  val diff = dot(features, effectiveCoefficientsVector) - label
/ labelStd + offset
  if (diff != 0) {
      val localGradientSumArray = gradientSumArray
      val localFeaturesStd = featuresStd
      features.foreachActive { (index, value) =>
         if (localFeaturesStd(index) != 0.0 && value != 0.0) {
           localGradientSumArray(index) += weight * diff * value
 / localFeaturesStd(index) // 见公式(11)
         }
```

```
      }
      lossSum += weight * diff * diff / 2.0   //见公式(3)
    }
    totalCnt += 1
    weightSum += weight
    this
  }

  def merge(other: LeastSquaresAggregator): this.type = {
      if (other.weightSum != 0) {
        totalCnt += other.totalCnt
        weightSum += other.weightSum
        lossSum += other.lossSum

        var i = 0
        val localThisGradientSumArray = this.gradientSumArray
        val localOtherGradientSumArray = other.gradientSumArray
        while (i < dim) {
          localThisGradientSumArray(i) += localOtherGradientSumArr
ay(i)
          i += 1
        }
      }
      this
    }
```

最后，根据下面的公式分别获取损失和梯度。

```
    def loss: Double = {
      lossSum / weightSum
    }

    def gradient: Vector = {
      val result = Vectors.dense(gradientSumArray.clone())
      scal(1.0 / weightSum, result)
      result
    }
```

- **3** 选择最优化方法

```
    val optimizer = if ($(elasticNetParam) == 0.0 || effectiveRe
gParam == 0.0) {
      new BreezeLBFGS[BDV[Double]]($(maxIter), 10, $(tol))
    } else {
      val standardizationParam = $(standardization)
      def effectiveL1RegFun = (index: Int) => {
        if (standardizationParam) {
          effectiveL1RegParam
        } else {
          // If `standardization` is false, we still standardize
 the data
          // to improve the rate of convergence; as a result, we
 have to
          // perform this reverse standardization by penalizing
each component
          // differently to get effectively the same objective f
unction when
          // the training dataset is not standardized.
          if (featuresStd(index) != 0.0) effectiveL1RegParam / f
eaturesStd(index) else 0.0
        }
      }
      new BreezeOWLQN[Int, BDV[Double]]($(maxIter), 10, effectiv
eL1RegFun, $(tol))
    }
```

如果没有正则化项或者只有L2正则化项，使用 `BreezeLBFGS` 来处理最优化问题，否则使用 `BreezeOWLQN` 。 `BreezeLBFGS` 和 `BreezeOWLQN` 的原理在相关章节会做具体介绍。

- **4** 获取结果，并做相应转换

```
val initialCoefficients = Vectors.zeros(numFeatures)
    val states = optimizer.iterations(new CachedDiffFunction(cos
tFun),
      initialCoefficients.asBreeze.toDenseVector)

    val (coefficients, objectiveHistory) = {
      val arrayBuilder = mutable.ArrayBuilder.make[Double]
      var state: optimizer.State = null
      while (states.hasNext) {
        state = states.next()
        arrayBuilder += state.adjustedValue
      }

      // 从标准空间转换到原来的空间
      val rawCoefficients = state.x.toArray.clone()
      var i = 0
      val len = rawCoefficients.length
      while (i < len) {
        rawCoefficients(i) *= { if (featuresStd(i) != 0.0) yStd
/ featuresStd(i) else 0.0 }
        i += 1
      }

      (Vectors.dense(rawCoefficients).compressed, arrayBuilder.r
esult())
    }

    // 系数收敛之后，intercept的计算可以通过封闭(`closed form`)的形式计
算出来，详细的讨论如下：
    // http://stats.stackexchange.com/questions/13617/how-is-the
-intercept-computed-in-glmnet
    val intercept = if ($(fitIntercept)) {
      yMean - dot(coefficients, Vectors.dense(featuresMean))
    } else {
      0.0
    }
```

朴素贝叶斯

1 介绍

朴素贝叶斯是一种构建分类器的简单方法。该分类器模型会给问题实例分配用特征值表示的类标签，类标签取自有限集合。它不是训练这种分类器的单一算法，而是一系列基于相同原理的算法：所有朴素贝叶斯分类器都假定样本每个特征与其他特征都不相关。 举个例子，如果一种水果其具有红，圆，直径大概3英寸等特征，该水果可以被判定为是苹果。尽管这些特征相互依赖或者有些特征由其他特征决定，然而朴素贝叶斯分类器认为这些属性在判定该水果是否为苹果的概率分布上独立的。

对于某些类型的概率模型，在有监督学习的样本集中能获取得非常好的分类效果。在许多实际应用中，朴素贝叶斯模型参数估计使用最大似然估计方法；换言之，在不用贝叶斯概率或者任何贝叶斯模型的情况下，朴素贝叶斯模型也能奏效。

尽管是带着这些朴素思想和过于简单化的假设，但朴素贝叶斯分类器在很多复杂的现实情形中仍能够取得相当好的效果。尽管如此，有论文证明更新的方法（如提升树和随机森林）的性能超过了贝叶斯分类器。

朴素贝叶斯分类器的一个优势在于只需要根据少量的训练数据估计出必要的参数（变量的均值和方差）。由于变量独立假设，只需要估计各个变量，而不需要确定整个协方差矩阵。

1.1 朴素贝叶斯的优缺点

- 优点：学习和预测的效率高，且易于实现；在数据较少的情况下仍然有效，可以处理多分类问题。
- 缺点：分类效果不一定很高，特征独立性假设会是朴素贝叶斯变得简单，但是会牺牲一定的分类准确率。

2 朴素贝叶斯概率模型

理论上，概率模型分类器是一个条件概率模型。

$$p(C|F_1,\dots,F_n)$$

独立的类别变量 `C` 有若干类别，条件依赖于若干特征变量 `F_1,F_2,...,F_n` 。但问题在于如果特征数量 `n` 较大或者每个特征能取大量值时，基于概率模型列出概率表变得不现实。所以我们修改这个模型使之变得可行。 贝叶斯定理有以下式子：

$$p(C|F_1,\dots,F_n) = \frac{p(C)\ p(F_1,\dots,F_n|C)}{p(F_1,\dots,F_n)}.$$

实际中，我们只关心分式中的分子部分，因为分母不依赖于 `C` 而且特征 `F_i` 的值是给定的，于是分母可以认为是一个常数。这样分子就等价于联合分布模型。 重复使用链式法则，可将该式写成条件概率的形式，如下所示：

$$\begin{aligned} p(C_k,x_1,\dots,x_n) &= p(x_1,\dots,x_n,C_k) \\ &= p(x_1|x_2,\dots,x_n,C_k)p(x_2,\dots,x_n,C_k) \\ &= p(x_1|x_2,\dots,x_n,C_k)p(x_2|x_3,\dots,x_n,C_k)p(x_3,\dots,x_n,C_k) \\ &= \dots \\ &= p(x_1|x_2,\dots,x_n,C_k)p(x_2|x_3,\dots,x_n,C_k)\dots p(x_{n-1}|x_n,C_k)p(x_n|C_k)p(C_k) \end{aligned}$$

现在“朴素”的条件独立假设开始发挥作用:假设每个特征 `F_i` 对于其他特征 `F_j` 是条件独立的。这就意味着

$$p(F_i|C,F_j) = p(F_i|C)$$

所以联合分布模型可以表达为

$$\begin{aligned} p(C|F_1,\dots,F_n) &\propto p(C)\ p(F_1|C)\ p(F_2|C)\ p(F_3|C)\ \cdots \\ &\propto p(C)\prod_{i=1}^{n} p(F_i|C). \end{aligned}$$

这意味着上述假设下，类变量 `C` 的条件分布可以表达为：

$$p(C|F_1,\dots,F_n) = \frac{1}{Z}p(C)\prod_{i=1}^{n} p(F_i|C)$$

其中 `Z` 是一个只依赖与 `F_1,...,F_n` 等的缩放因子，当特征变量的值已知时是一个常数。

从概率模型中构造分类器

讨论至此为止我们导出了独立分布特征模型，也就是朴素贝叶斯概率模型。朴素贝叶斯分类器包括了这种模型和相应的决策规则。一个普通的规则就是选出最有可能的那个：这就是大家熟知的最大后验概率（ `MAP` ）决策准则。相应的分类器便是如下定义的公式：

$$\text{classify}(f_1, \ldots, f_n) = \underset{c}{\operatorname{argmax}}\ p(C=c) \prod_{i=1}^{n} p(F_i = f_i | C = c).$$

3 参数估计

所有的模型参数都可以通过训练集的相关频率来估计。常用方法是概率的最大似然估计。类的先验概率 `P(C)` 可以通过假设各类等概率来计算 `（先验概率 = 1 / (类的数量)）` ，或者通过训练集的各类样本出现的次数来估计 `（A类先验概率=（A类样本的数量）/(样本总数)）` 。

对于类条件概率 `P(X|c)` 来说，直接根据样本出现的频率来估计会很困难。在现实应用中样本空间的取值往往远远大于训练样本数，也就是说，很多样本取值在训练集中根本没有出现，直接使用频率来估计 `P(x|c)` 不可行，因为"未被观察到"和"出现概率为零"是不同的。 为了估计特征的分布参数，我们要先假设训练集数据满足某种分布或者非参数模型。

这种假设称为朴素贝叶斯分类器的事件模型（ `event model` ）。对于离散的特征数据（例如文本分类中使用的特征），多元分布和伯努利分布比较流行。

3.1 高斯朴素贝叶斯

如果要处理的是连续数据，一种通常的假设是这些连续数值服从高斯分布。例如，假设训练集中有一个连续属性 `x` 。我们首先对数据根据类别分类，然后计算每个类别中 `x` 的均值和方差。令 `mu_c` 表示为 `x` 在 `c` 类上的均值，令 `sigma^2_c` 为 `x` 在 `c` 类上的方差。在给定类中某个值的概率 `P(x=v|c)` ，可以通过将 `v` 表示为均值为 `mu_c` ，方差为 `sigma^2_c` 的正态分布计算出来。

$$p(x=v|c) = \frac{1}{\sqrt{2\pi\sigma_c^2}} e^{-\frac{(v-\mu_c)^2}{2\sigma_c^2}}$$

处理连续数值问题的另一种常用的技术是通过离散化连续数值的方法。通常，当训练样本数量较少或者是精确的分布已知时，通过概率分布的方法是一种更好的选择。 在大量样本的情形下离散化的方法表现更优，因为大量的样本可以学习到数据的分布。由于朴素贝叶斯是一种典型的用到大量样本的方法（越大计算量的模型可以产生越高的分类精确度），所以朴素贝叶斯方法都用到离散化方法，而不是概率分布估计的方法。

3.2 多元朴素贝叶斯

在多元事件模型中，样本（特征向量）表示特定事件发生的次数。用 `p_i` 表示事件 `i` 发生的概率。特征向量 `X=(x_1,x_2,...,x_n)` 是一个 `histogram` ，其中 `x_i` 表示事件 `i` 在特定的对象中被观察到的次数。事件模型通常用于文本分类。相应的 `x_i` 表示词 `i` 在单个文档中出现的次数。 `X` 的似然函数如下所示：

$$p(\mathbf{x}|C_k) = \frac{(\sum_i x_i)!}{\prod_i x_i!} \prod_i {p_{ki}}^{x_i}$$

当用对数空间表达时，多元朴素贝叶斯分类器变成了线性分类器。

$$\begin{aligned} \log p(C_k|\mathbf{x}) &\propto \log \left(p(C_k) \prod_{i=1}^{n} {p_{ki}}^{x_i} \right) \\ &= \log p(C_k) + \sum_{i=1}^{n} x_i \cdot \log p_{ki} \\ &= b + \mathbf{w}_k^\top \mathbf{x} \end{aligned}$$

如果一个给定的类和特征值在训练集中没有一起出现过，那么基于频率的估计下该概率将为0。这将是一个问题。因为与其他概率相乘时将会把其他概率的信息统统去除。所以常常要求要对每个小类样本的概率估计进行修正，以保证不会出现有为0的概率出现。常用到的平滑就是加1平滑（也称拉普拉斯平滑）。

根据参考文献【2】，我们以文本分类的训练和测试为例子来介绍多元朴素贝叶斯的训练和测试过程。如下图所示。

```
TrainMultinomialNB(ℂ, 𝔻)
 1  V ← ExtractVocabulary(𝔻)
 2  N ← CountDocs(𝔻)
 3  for each c ∈ ℂ
 4  do N_c ← CountDocsInClass(𝔻, c)
 5      prior[c] ← N_c / N
 6      text_c ← ConcatenateTextOfAllDocsInClass(𝔻, c)
 7      for each t ∈ V
 8      do T_ct ← CountTokensOfTerm(text_c, t)
 9      for each t ∈ V
10      do condprob[t][c] ← (T_ct + 1) / Σ_t'(T_ct' + 1)
11  return V, prior, condprob

ApplyMultinomialNB(ℂ, V, prior, condprob, d)
 1  W ← ExtractTokensFromDoc(V, d)
 2  for each c ∈ ℂ
 3  do score[c] ← log prior[c]
 4      for each t ∈ W
 5      do score[c] += log condprob[t][c]
 6  return arg max_{c∈ℂ} score[c]
```

这里的`CondProb[t][c]`即上文中的`P(x|C)`。`T_ct`表示类别为`c`的文档中`t`出现的次数。`+1`就是平滑手段。

3.3 伯努利朴素贝叶斯

在多变量伯努利事件模型中，特征是独立的二值变量。和多元模型一样，这个模型在文本分类中也非常流行。它的似然函数如下所示。

$$p(\mathbf{x}|C_k) = \prod_{i=1}^{n} p_{ki}^{x_i}(1-p_{ki})^{(1-x_i)}$$

其中`p_ki`表示类别`C_k`生成`term` `w_i`的概率。这个模型通常用于短文本分类。

根据参考文献【2】，我们以文本分类的训练和测试为例子来介绍多元朴素贝叶斯的训练和测试过程。如下图所示。

TrainBernoulliNB($\mathbb{C}, \mathbb{D}$)
1 $V \leftarrow$ ExtractVocabulary($\mathbb{D}$)
2 $N \leftarrow$ CountDocs($\mathbb{D}$)
3 **for each** $c \in \mathbb{C}$
4 **do** $N_c \leftarrow$ CountDocsInClass($\mathbb{D}, c$)
5 $\quad$ $prior[c] \leftarrow N_c / N$
6 $\quad$ **for each** $t \in V$
7 $\quad$ **do** $N_{ct} \leftarrow$ CountDocsInClassContainingTerm($\mathbb{D}, c, t$)
8 $\quad\quad$ $condprob[t][c] \leftarrow (N_{ct} + 1)/(N_c + 2)$
9 **return** $V, prior, condprob$

ApplyBernoulliNB($\mathbb{C}, V, prior, condprob, d$)
1 $V_d \leftarrow$ ExtractTermsFromDoc(V, d)
2 **for each** $c \in \mathbb{C}$
3 **do** $score[c] \leftarrow \log prior[c]$
4 $\quad$ **for each** $t \in V$
5 $\quad$ **do if** $t \in V_d$
6 $\quad\quad$ **then** $score[c] \mathrel{+}= \log condprob[t][c]$
7 $\quad\quad$ **else** $score[c] \mathrel{+}= \log(1 - condprob[t][c])$
8 **return** $\arg\max_{c \in \mathbb{C}} score[c]$

► **Figure 13.1** NB algorithm (Bernoulli model): Training and testing. The add-one smoothing in Line 8 (top) is in analogy to Equation 119 with $B = 2$.

4 源码分析

`MLlib` 中实现了多元朴素贝叶斯和伯努利朴素贝叶斯。下面先看看朴素贝叶斯的使用实例。

4.1 实例

```
import org.apache.spark.mllib.classification.{NaiveBayes, NaiveBayesModel}
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
//读取并处理数据
val data = sc.textFile("data/mllib/sample_naive_bayes_data.txt")
val parsedData = data.map { line =>
  val parts = line.split(',')
  LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}
// 切分数据为训练数据和测试数据
val splits = parsedData.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0)
val test = splits(1)
//训练模型
val model = NaiveBayes.train(training, lambda = 1.0, modelType = "multinomial")
//测试数据
val predictionAndLabel = test.map(p => (model.predict(p.features), p.label))
val accuracy = 1.0 * predictionAndLabel.filter(x => x._1 == x._2).count() / test.count()
```

4.2 训练模型

从上文的原理分析我们可以知道，朴素贝叶斯模型的训练过程就是获取概率 `p(C)` 和 `p(F|C)` 的过程。根据 `MLlib` 的源码，我们可以将训练过程分为两步。 第一步是聚合计算每个标签对应的 `term` 的频率，第二步是迭代计算 `p(C)` 和 `p(F|C)` 。

- **1** 计算每个标签对应的 `term` 的频率

```
val aggregated = data.map(p => (p.label, p.features)).combineByK
ey[(Long, DenseVector)](
      createCombiner = (v: Vector) => {
        if (modelType == Bernoulli) {
          requireZeroOneBernoulliValues(v)
        } else {
          requireNonnegativeValues(v)
        }
        (1L, v.copy.toDense)
      },
      mergeValue = (c: (Long, DenseVector), v: Vector) => {
        requireNonnegativeValues(v)
        //c._2 = v*1 + c._2
        BLAS.axpy(1.0, v, c._2)
        (c._1 + 1L, c._2)
      },
      mergeCombiners = (c1: (Long, DenseVector), c2: (Long, Dens
eVector)) => {
        BLAS.axpy(1.0, c2._2, c1._2)
        (c1._1 + c2._1, c1._2)
      }
    //根据标签进行排序
    ).collect().sortBy(_._1)
```

这里我们需要先了解 `createCombiner` 函数的作用。`createCombiner` 的作用是将原 `RDD` 中的 `Vector` 类型转换为 `(long,Vector)` 类型。

如果 `modelType` 为 `Bernoulli` ，那么 `v` 中包含的值只能为0或者1。如果 `modelType` 为 `multinomial` ，那么 `v` 中包含的值必须大于0。

```
//值非负
val requireNonnegativeValues: Vector => Unit = (v: Vector) => {
      val values = v match {
        case sv: SparseVector => sv.values
        case dv: DenseVector => dv.values
      }
      if (!values.forall(_ >= 0.0)) {
        throw new SparkException(s"Naive Bayes requires nonnegat
ive feature values but found $v.")
      }
}
//值为0或者1
val requireZeroOneBernoulliValues: Vector => Unit = (v: Vector)
=> {
      val values = v match {
        case sv: SparseVector => sv.values
        case dv: DenseVector => dv.values
      }
      if (!values.forall(v => v == 0.0 || v == 1.0)) {
        throw new SparkException(
          s"Bernoulli naive Bayes requires 0 or 1 feature values
 but found $v.")
      }
}
```

`mergeValue` 函数的作用是将新来的 `Vector` 累加到已有向量中，并更新词率。 `mergeCombiners` 则是合并不同分区的 `(long,Vector)` 数据。 通过这个函数，我们就找到了每个标签对应的词频率，并得到了标签对应的所有文档的累加向量。

- **2** 迭代计算 `p(C)` 和 `p(F|C)`

```
//标签数
val numLabels = aggregated.length
//文档数
var numDocuments = 0L
aggregated.foreach { case (_, (n, _)) =>
  numDocuments += n
}
```

```
//特征维数
val numFeatures = aggregated.head match { case (_, (_, v)) => v.
size }
val labels = new Array[Double](numLabels)
//表示logP(C)
val pi = new Array[Double](numLabels)
//表示logP(F|C)
val theta = Array.fill(numLabels)(new Array[Double](numFeatures)
)
val piLogDenom = math.log(numDocuments + numLabels * lambda)
var i = 0
aggregated.foreach { case (label, (n, sumTermFreqs)) =>
      labels(i) = label
      //训练步骤的第5步
      pi(i) = math.log(n + lambda) - piLogDenom
      val thetaLogDenom = modelType match {
        case Multinomial => math.log(sumTermFreqs.values.sum + n
umFeatures * lambda)
        case Bernoulli => math.log(n + 2.0 * lambda)
        case _ =>
          // This should never happen.
          throw new UnknownError(s"Invalid modelType: $modelType
.")
      }
      //训练步骤的第6步
      var j = 0
      while (j < numFeatures) {
        theta(i)(j) = math.log(sumTermFreqs(j) + lambda) - theta
LogDenom
        j += 1
      }
      i += 1
    }
```

这段代码计算上文提到的 `p(C)` 和 `p(F|C)` 。这里的 `lambda` 表示平滑因子，一般情况下，我们将它设置为1。代码中， `p(c_i)=log(n+lambda)/(numDocs+numLabels*lambda)` ，这对应上文训练过程的第5步 `prior(c)=N_c/N` 。

根据 `modelType` 类型的不同， `p(F|C)` 的实现则不同。当 `modelType` 为 `Multinomial` 时， `P(F|C)=T_ct/sum(T_ct)` ，这里 `sum(T_ct)=sumTermFreqs.values.sum + numFeatures * lambda` 。这对应多元朴素贝叶斯训练过程的第10步。 当 `modelType` 为 `Bernoulli` 时， `P(F|C)=(N_ct+lambda)/(N_c+2*lambda)` 。这对应伯努利贝叶斯训练算法的第8行。

需要注意的是，代码中的所有计算都是取对数计算的。

4.3 预测数据

```
override def predict(testData: Vector): Double = {
    modelType match {
      case Multinomial =>
        labels(multinomialCalculation(testData).argmax)
      case Bernoulli =>
        labels(bernoulliCalculation(testData).argmax)
    }
}
```

预测也是根据 `modelType` 的不同作不同的处理。当 `modelType` 为 `Multinomial` 时，调用 `multinomialCalculation` 函数。

```
private def multinomialCalculation(testData: Vector) = {
    val prob = thetaMatrix.multiply(testData)
    BLAS.axpy(1.0, piVector, prob)
    prob
  }
```

这里的 `thetaMatrix` 和 `piVector` 即上文中训练得到的 `P(F|C)` 和 `P(C)` ，根据 `P(C|F)=P(F|C)*P(C)` 即可以得到预测数据归属于某类别的概率。 注意，这些概率都是基于对数结果计算的。

当 `modelType` 为 `Bernoulli` 时，实现代码略有不同。

```
private def bernoulliCalculation(testData: Vector) = {
    testData.foreachActive((_, value) =>
      if (value != 0.0 && value != 1.0) {
        throw new SparkException(
          s"Bernoulli naive Bayes requires 0 or 1 feature values
 but found $testData.")
      }
    )
    val prob = thetaMinusNegTheta.get.multiply(testData)
    BLAS.axpy(1.0, piVector, prob)
    BLAS.axpy(1.0, negThetaSum.get, prob)
    prob
  }
```

当词在训练数据中出现与否处理的过程不同。见伯努利模型测试过程。这里用矩阵和向量的操作来实现这个过程，需要仔细体会。

```
 private val (thetaMinusNegTheta, negThetaSum) = modelType match
 {
    case Multinomial => (None, None)
    case Bernoulli =>
      val negTheta = thetaMatrix.map(value => math.log(1.0 - mat
h.exp(value)))
      val ones = new DenseVector(Array.fill(thetaMatrix.numCols){
1.0})
      val thetaMinusNegTheta = thetaMatrix.map { value =>
        value - math.log(1.0 - math.exp(value))
      }
      (Option(thetaMinusNegTheta), Option(negTheta.multiply(ones
)))
    case _ =>
      // This should never happen.
      throw new UnknownError(s"Invalid modelType: $modelType.")
  }
```

这里 `math.exp(value)` 将对数概率恢复成真实的概率。

参考文献

【1】朴素贝叶斯分类器

【2】Naive Bayes text classification

【3】The Bernoulli model

决策树

1 决策树理论

1.1 什么是决策树

所谓决策树，顾名思义，是一种树，一种依托于策略抉择而建立起来的树。机器学习中，决策树是一个预测模型；他代表的是对象属性与对象值之间的一种映射关系。树中每个节点表示某个对象，而每个分叉路径则代表的某个可能的属性值，从根节点到叶节点所经历的路径对应一个判定测试序列。决策树仅有单一输出，若欲有复数输出，可以建立独立的决策树以处理不同输出。

1.2 决策树学习流程

决策树学习的主要目的是为了产生一棵泛化能力强的决策树。其基本流程遵循简单而直接的“分而治之”的策略。它的流程实现如下所示：

```
输入：训练集 D={(x_1,y_1),(x_2,y_2),...,(x_m,y_m)};
      属性集 A={a_1,a_2,...,a_d}
过程：函数GenerateTree(D,A)
1: 生成节点node；
2: if D中样本全属于同一类别C then
3:     将node标记为C类叶节点，并返回
4: end if
5: if A为空 OR D中样本在A上取值相同 then
6:     将node标记为叶节点，其类别标记为D中样本数量最多的类，并返回
7: end if
8: 从A中选择最优划分属性 a*；    //每个属性包含若干取值，这里假设有v个取值
9: for a* 的每个值a*_v do
10:     为node生成一个分支，令D_v表示D中在a*上取值为a*_v的样本子集；
11:     if D_v 为空 then
12:         将分支节点标记为叶节点，其类别标记为D中样本最多的类，并返回
13:     else
14:         以GenerateTree(D_v,A\{a*})为分支节点
15:     end if
16: end for
```

决策树的生成是一个递归的过程。有三种情况会导致递归的返回：（1）当前节点包含的样本全属于同一个类别。（2）当前属性值为空，或者所有样本在所有属性上取相同的值。（3）当前节点包含的样本集合为空。

在第（2）中情形下，我们把当前节点标记为叶节点，并将其类别设定为该节点所含样本最多的类别；在第（3）中情形下，同样把当前节点标记为叶节点，但是将其类别设定为其父节点所含样本最多的类别。这两种处理实质不同，前者利用当前节点的后验分布，后者则把父节点的样本分布作为当前节点的先验分布。

1.3 决策树的构造

构造决策树的关键步骤是分裂属性（即确定属性的不同取值，对应上面流程中的 `a_v` ）。所谓分裂属性就是在某个节点处按照某一属性的不同划分构造不同的分支，其目标是让各个分裂子集尽可能地“纯”。尽可能“纯”就是尽量让一个分裂子集中待分类项属于同一类别。分裂属性分为三种不同的情况：

- 1、属性是离散值且不要求生成二叉决策树。此时用属性的每一个划分作为一个分支。

- 2、属性是离散值且要求生成二叉决策树。此时使用属性划分的一个子集进行测试，按照“属于此子集”和“不属于此子集”分成两个分支。
- 3、属性是连续值。此时确定一个值作为分裂点 `split_point` ，按照 `>split_point` 和 `<=split_point` 生成两个分支。

1.4 划分选择

在决策树算法中，如何选择最优划分属性是最关键的一步。一般而言，随着划分过程的不断进行，我们希望决策树的分支节点所包含的样本尽可能属于同一类别，即节点的“纯度(purity)”越来越高。 有几种度量样本集合纯度的指标。在 `MLlib` 中，信息熵和基尼指数用于决策树分类，方差用于决策树回归。

1.4.1 信息熵

信息熵是度量样本集合纯度最常用的一种指标，假设当前样本集合 `D` 中第 `k` 类样本所占的比例为 `p_k` ，则 `D` 的信息熵定义为：

$$\mathrm{Ent}(D) = -\sum_{k=1}^{|y|} p_k \log_2 p_k$$

`Ent(D)` 的值越小，则 `D` 的纯度越高。

1.4.2 基尼系数

采用和上式相同的符号，基尼系数可以用来度量数据集 `D` 的纯度。

$$\mathrm{Gini}(D) = \sum_{k=1}^{|y|} \sum_{k' \neq k} p_k p_{k'} = 1 - \sum_{k=1}^{|y|} p_k^2$$

直观来说， `Gini(D)` 反映了从数据集 `D` 中随机取样两个样本，其类别标记不一致的概率。因此， `Gini(D)` 越小，则数据集 `D` 的纯度越高。

1.4.3 方差

`MLlib` 中使用方差来度量纯度。如下所示

$$\mathrm{Var}(D) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \frac{1}{N}\sum_{i=1}^{N} y_i)$$

1.4.4 信息增益

假设切分大小为 `N` 的数据集 `D` 为两个数据集 `D_left` 和 `D_right` ，那么信息增益可以表示为如下的形式。

$$\mathrm{IG}(\mathrm{D},\mathrm{s}) = \mathrm{Impurity}(\mathrm{D}) - \frac{N_{left}}{N} Impurity(D_{left}) - \frac{N_{right}}{N} Impurity(D_{right})$$

一般情况下，信息增益越大，则意味着使用属性 `a` 来进行划分所获得的纯度提升越大。因此我们可以用信息增益来进行决策树的划分属性选择。即流程中的第8步。

1.5 决策树的优缺点

决策树的优点：

- 1 决策树易于理解和解释；
- 2 能够同时处理数据型和类别型属性；
- 3 决策树是一个白盒模型，给定一个观察模型，很容易推出相应的逻辑表达式；
- 4 在相对较短的时间内能够对大型数据作出效果良好的结果；
- 5 比较适合处理有缺失属性值的样本。

决策树的缺点：

- 1 对那些各类别数据量不一致的数据，在决策树种，信息增益的结果偏向那些具有更多数值的特征；
- 2 容易过拟合；
- 3 忽略了数据集中属性之间的相关性。

2 实例与源码分析

2.1 实例

下面的例子用于分类。

```
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.util.MLUtils
// Load and parse the data file.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_
data.txt")
// Split the data into training and test sets (30% held out for
testing)
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
// Train a DecisionTree model.
//  Empty categoricalFeaturesInfo indicates all features are con
tinuous.
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "gini"
val maxDepth = 5
val maxBins = 32
val model = DecisionTree.trainClassifier(trainingData, numClasse
s, categoricalFeaturesInfo,
  impurity, maxDepth, maxBins)
// Evaluate model on test instances and compute test error
val labelAndPreds = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count().to
Double / testData.count()
println("Test Error = " + testErr)
println("Learned classification tree model:\n" + model.toDebugSt
ring)
```

下面的例子用于回归。

```
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.util.MLUtils
// Load and parse the data file.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_
data.txt")
// Split the data into training and test sets (30% held out for
testing)
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
// Train a DecisionTree model.
//  Empty categoricalFeaturesInfo indicates all features are con
tinuous.
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "variance"
val maxDepth = 5
val maxBins = 32
val model = DecisionTree.trainRegressor(trainingData, categorica
lFeaturesInfo, impurity,
  maxDepth, maxBins)
// Evaluate model on test instances and compute test error
val labelsAndPredictions = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testMSE = labelsAndPredictions.map{ case (v, p) => math.pow(
v - p, 2) }.mean()
println("Test Mean Squared Error = " + testMSE)
println("Learned regression tree model:\n" + model.toDebugString
)
```

2.2 源码分析

在 `MLlib` 中，决策树的实现和随机森林的实现是在一起的。随机森林实现中，当树的个数为1时，它的实现即为决策树的实现。

```
def run(input: RDD[LabeledPoint]): DecisionTreeModel = {
    //树个数为1
    val rf = new RandomForest(strategy, numTrees = 1, featureSub
setStrategy = "all", seed = 0)
    val rfModel = rf.run(input)
    rfModel.trees(0)
  }
```

这里的 `strategy` 是 `Strategy` 的实例，它包含如下信息：

```
/**
 * Stores all the configuration options for tree construction
 * @param algo  Learning goal.  Supported:
 *              [[org.apache.spark.mllib.tree.configuration.Algo
.Classification]],
 *              [[org.apache.spark.mllib.tree.configuration.Algo
.Regression]]
 * @param impurity Criterion used for information gain calculati
on.
 *                 Supported for Classification: [[org.apache.sp
ark.mllib.tree.impurity.Gini]],
 *                  [[org.apache.spark.mllib.tree.impurity.Entro
py]].
 *                 Supported for Regression: [[org.apache.spark.
mllib.tree.impurity.Variance]].
 * @param maxDepth Maximum depth of the tree.
 *                 E.g., depth 0 means 1 leaf node; depth 1 mean
s 1 internal node + 2 leaf nodes.
 * @param numClasses Number of classes for classification.
 *                                    (Ignored for regression.)
 *                                    Default value is 2 (binary
 classification).
 * @param maxBins Maximum number of bins used for discretizing c
ontinuous features and
 *                for choosing how to split on features at each
node.
 *                More bins give higher granularity.
 * @param quantileCalculationStrategy Algorithm for calculating
quantiles.  Supported:
```

```
 *                              [[org.apache.spark.mllib.tree.con
figuration.QuantileStrategy.Sort]]
 * @param categoricalFeaturesInfo A map storing information abou
t the categorical variables and the
 *                                number of discrete values they
 take. For example, an entry (n ->
 *                                k) implies the feature n is ca
tegorical with k categories 0,
 *                                1, 2, ... , k-1. It's importan
t to note that features are
 *                                zero-indexed.
 * @param minInstancesPerNode Minimum number of instances each c
hild must have after split.
 *                            Default value is 1. If a split cau
se left or right child
 *                            to have less than minInstancesPerN
ode,
 *                            this split will not be considered
as a valid split.
 * @param minInfoGain Minimum information gain a split must get.
 Default value is 0.0.
 *                    If a split has less information gain than
minInfoGain,
 *                    this split will not be considered as a val
id split.
 * @param maxMemoryInMB Maximum memory in MB allocated to histog
ram aggregation. Default value is
 *                      256 MB.
 * @param subsamplingRate Fraction of the training data used for
 learning decision tree.
 * @param useNodeIdCache If this is true, instead of passing tre
es to executors, the algorithm will
 *                      maintain a separate RDD of node Id cache
 for each row.
 * @param checkpointInterval How often to checkpoint when the no
de Id cache gets updated.
 *                           E.g. 10 means that the cache will g
et checkpointed every 10 updates. If
 *                           the checkpoint directory is not set
 in
```

```
 *                            [[org.apache.spark.SparkContext]],
this setting is ignored.
 */
class Strategy @Since("1.3.0") (
    @Since("1.0.0") @BeanProperty var algo: Algo,//选择的算法，有分类和回归两种选择
    @Since("1.0.0") @BeanProperty var impurity: Impurity,//纯度有熵、基尼系数、方差三种选择
    @Since("1.0.0") @BeanProperty var maxDepth: Int,//树的最大深度
    @Since("1.2.0") @BeanProperty var numClasses: Int = 2,//分类数

    @Since("1.0.0") @BeanProperty var maxBins: Int = 32,//最大子树个数
    @Since("1.0.0") @BeanProperty var quantileCalculationStrategy: QuantileStrategy = Sort,
    //保存类别变量以及相应的离散值。一个entry (n ->k) 表示特征n属于k个类别，分别是0,1,...,k-1
    @Since("1.0.0") @BeanProperty var categoricalFeaturesInfo: Map[Int, Int] = Map[Int, Int](),
    @Since("1.2.0") @BeanProperty var minInstancesPerNode: Int = 1,
    @Since("1.2.0") @BeanProperty var minInfoGain: Double = 0.0,
    @Since("1.0.0") @BeanProperty var maxMemoryInMB: Int = 256,
    @Since("1.2.0") @BeanProperty var subsamplingRate: Double = 1,
    @Since("1.2.0") @BeanProperty var useNodeIdCache: Boolean = false,
    @Since("1.2.0") @BeanProperty var checkpointInterval: Int = 10) extends Serializable
```

决策树的实现我们在随机森林专题介绍。这里我们只需要知道，当随机森林的树个数为1时，它即为决策树， 并且此时，树的训练所用的特征是全部特征，而不是随机选择的部分特征。即 `featureSubsetStrategy = "all"` 。

集成学习

集成学习通过构建并结合多个学习器来完成学习任务，有时也被称为多分类器系统。集成学习通过将多个学习器进行结合，常可获得比单一学习器显著优越的泛化能力。

根据个体学习器的生成方式，目前的集成学习方法大致可以分为两大类。即个体学习器之间存在强依赖性，必须串行生成的序列化方法以及个体学习器之间不存在强依赖性，可同时生成的并行化方法。 前者的代表是`Boosting`，后者的代表是`Bagging`和随机森林。后面的随机森林章节会详细介绍`Bagging`和随机森林；梯度提升树章节会详细介绍`Boosting`和梯度提升树。

随机森林

1 Bagging

`Bagging` 采用自助采样法(`bootstrap sampling`)采样数据。给定包含 `m` 个样本的数据集，我们先随机取出一个样本放入采样集中，再把该样本放回初始数据集，使得下次采样时，样本仍可能被选中， 这样，经过 `m` 次随机采样操作，我们得到包含 `m` 个样本的采样集。

按照此方式，我们可以采样出 `T` 个含 `m` 个训练样本的采样集，然后基于每个采样集训练出一个基本学习器，再将这些基本学习器进行结合。这就是 `Bagging` 的一般流程。在对预测输出进行结合时， `Bagging` 通常使用简单投票法， 对回归问题使用简单平均法。若分类预测时，出现两个类收到同样票数的情形，则最简单的做法是随机选择一个，也可以进一步考察学习器投票的置信度来确定最终胜者。

`Bagging` 的算法描述如下图所示。

输入：训练集 $D=\{(x_1,y_1),(x_2,y_2),\ldots,(x_m,y_m)\}$;
　　　基础学习算法ζ
　　　训练次数 T
过程：
1：for t=1,2,...,T do
2：　　$h_t=\zeta(D,D_{bs})$
3：end for
输出：$H(x)=\arg max_{y\in Y}\sum_{t=1}^{T} I(h_t(x)=y)$

2 随机森林

随机森林是 `Bagging` 的一个扩展变体。随机森林在以决策树为基学习器构建 `Bagging` 集成的基础上，进一步在决策树的训练过程中引入了随机属性选择。具体来讲，传统决策树在选择划分属性时， 在当前节点的属性集合（假设有 `d` 个属性）中选择一个最优属性；而在随机森林中，对基决策树的每个节点，先从该节点的属性集合中随机选择一个包含 `k` 个属性的子集，然后再从这个子集中选择一个最优属性用于划分。 这里的参数 `k` 控制了随机性的引入程度。若令 `k=d` ，则

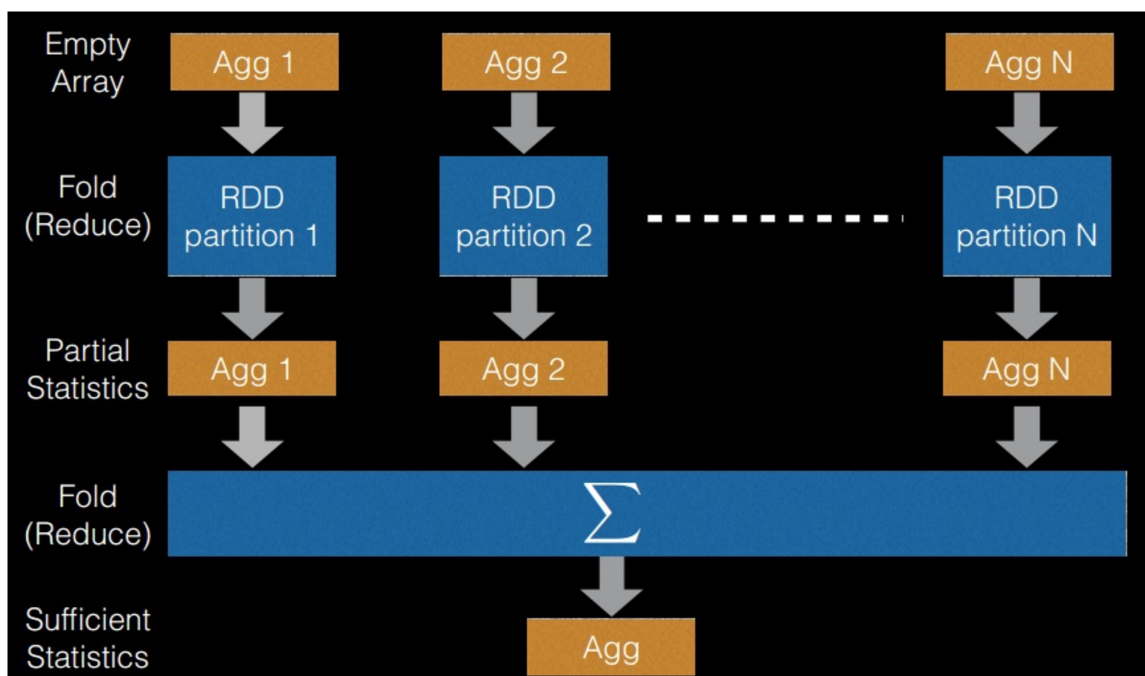
基决策树的构建与传统决策树相同；若令 `k=1` ，则是随机选择一个属性用于划分。在 `MLlib` 中，有两种选择用于分类，即 `k=log2(d)` 、 `k=sqrt(d)` ； 一种选择用于回归，即 `k=1/3d` 。在源码分析中会详细介绍。

可以看出，随机森林对 `Bagging` 只做了小改动，但是与 `Bagging` 中基学习器的“多样性”仅仅通过样本扰动（通过对初始训练集采样）而来不同，随机森林中基学习器的多样性不仅来自样本扰动，还来自属性扰动。 这使得最终集成的泛化性能可通过个体学习器之间差异度的增加而进一步提升。

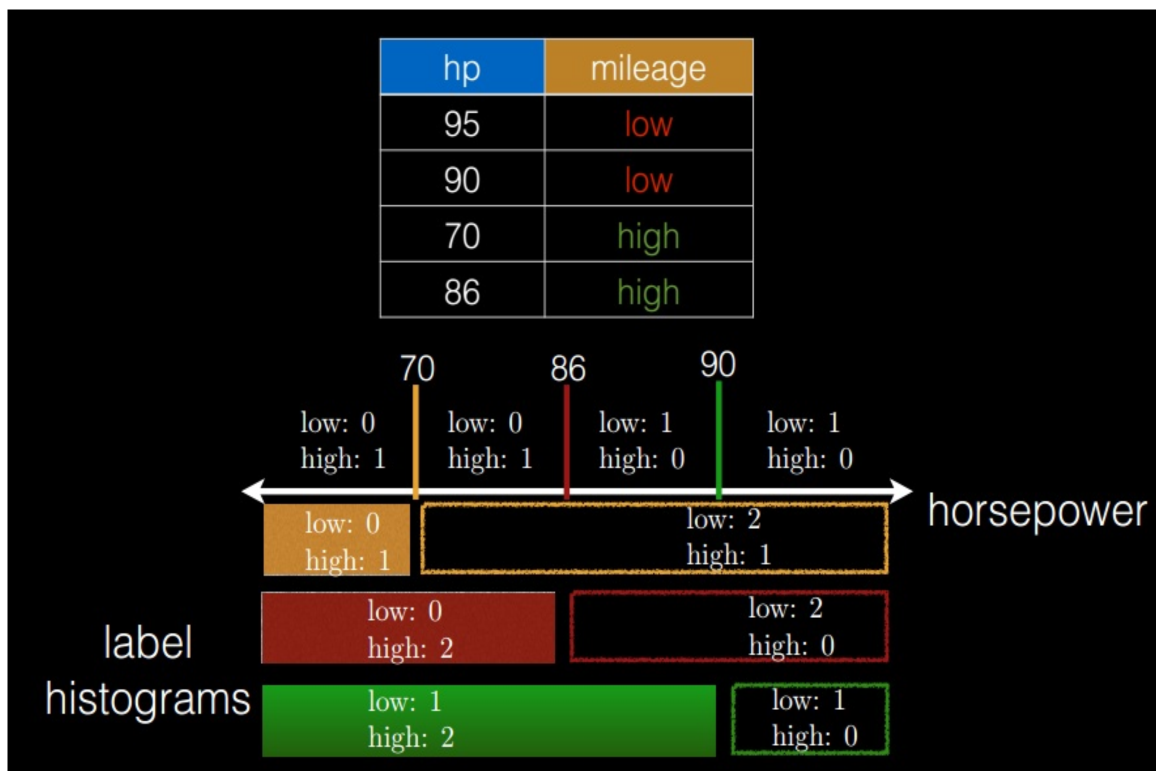
3 随机森林在分布式环境下的优化策略

随机森林算法在单机环境下很容易实现，但在分布式环境下特别是在 `Spark` 平台上，传统单机形式的迭代方式必须要进行相应改进才能适用于分布式环境 ，这是因为在分布式环境下，数据也是分布式的，算法设计不得当会生成大量的 `IO` 操作，例如频繁的网络数据传输，从而影响算法效率。 因此，在 `Spark` 上进行随机森林算法的实现，需要进行一定的优化， `Spark` 中的随机森林算法主要实现了三个优化策略：

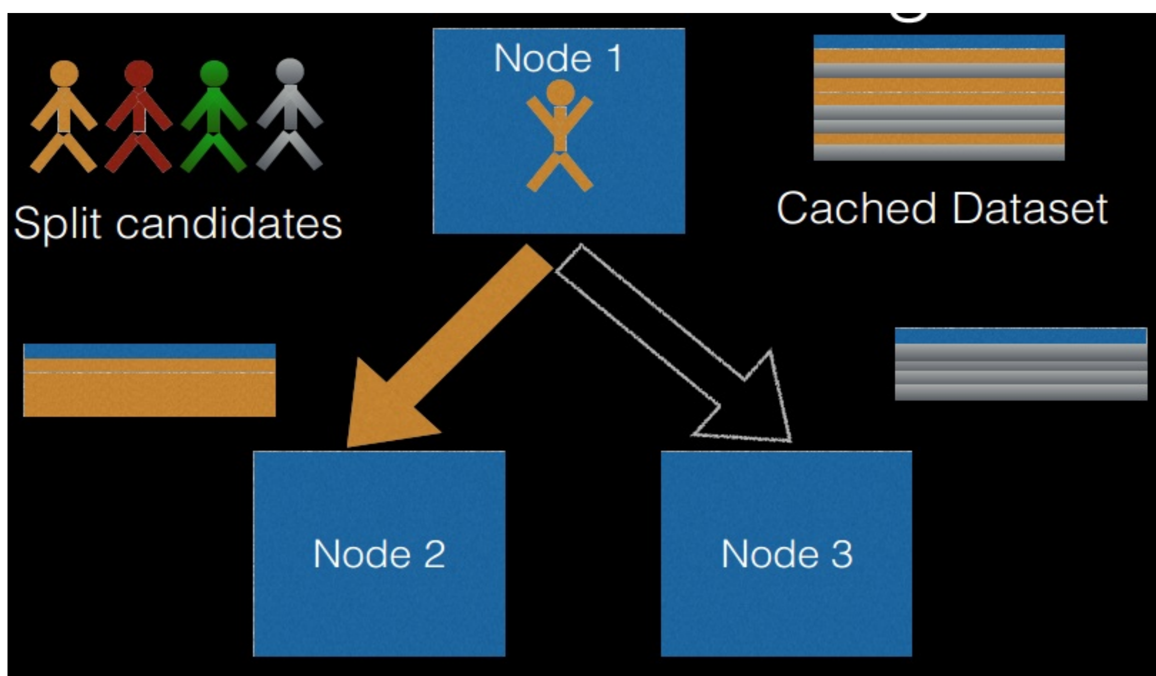
- 切分点抽样统计，如下图所示。在单机环境下的决策树对连续变量进行切分点选择时，一般是通过对特征点进行排序，然后取相邻两个数之间的点作为切分点，这在单机环境下是可行的，但如果在分布式环境下如此操作的话， 会带来大量的网络传输操作，特别是当数据量达到 `PB` 级时，算法效率将极为低下。为避免该问题， `Spark` 中的随机森林在构建决策树时，会对各分区采用一定的子特征策略进行抽样，然后生成各个分区的统计数据，并最终得到切分点。(从源代码里面看，是先对样本进行抽样，然后根据抽样样本值出现的次数进行排序，然后再进行切分)。



- 特征装箱（`Binning`），如下图所示。决策树的构建过程就是对特征的取值不断进行划分的过程，对于离散的特征，如果有`M`个值，最多有`2^(M-1) - 1`个划分。如果值是有序的，那么就最多`M-1`个划分。比如年龄特征，有老，中，少3个值，如果无序有`2^2-1=3`个划分，即`老|中，少；老，中|少；老，少|中`。；如果是有序的，即按老，中，少的序，那么只有`m-1`个，即2种划分，`老|中，少；老，中|少`。对于连续的特征，其实就是进行范围划分，而划分的点就是`split`（切分点），划分出的区间就是`bin`。对于连续特征，理论上`split`是无数的，在分布环境下不可能取出所有的值，因此它采用的是切点抽样统计方法。



- 逐层训练（`level-wise training`），如下图所示。单机版本的决策树生成过程是通过递归调用（本质上是深度优先）的方式构造树，在构造树的同时，需要移动数据，将同一个子节点的数据移动到一起。此方法在分布式数据结构上无法有效的执行，而且也无法执行，因为数据太大，无法放在一起，所以在分布式环境下采用的策略是逐层构建树节点（本质上是广度优先），这样遍历所有数据的次数等于所有树中的最大层数。每次遍历时，只需要计算每个节点所有切分点统计参数，遍历完后，根据节点的特征划分，决定是否切分，以及如何切分。



4 使用实例

下面的例子用于分类。

```
import org.apache.spark.mllib.tree.RandomForest
import org.apache.spark.mllib.tree.model.RandomForestModel
import org.apache.spark.mllib.util.MLUtils
// Load and parse the data file.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_
data.txt")
// Split the data into training and test sets (30% held out for
testing)
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
// Train a RandomForest model.
// 空的类别特征信息表示所有的特征都是连续的.
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val numTrees = 3 // Use more in practice.
val featureSubsetStrategy = "auto" // Let the algorithm choose.
val impurity = "gini"
val maxDepth = 4
val maxBins = 32
val model = RandomForest.trainClassifier(trainingData, numClasse
s, categoricalFeaturesInfo,
  numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins)
// Evaluate model on test instances and compute test error
val labelAndPreds = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDo
uble / testData.count()
println("Test Error = " + testErr)
println("Learned classification forest model:\n" + model.toDebug
String)
```

下面的例子用于回归。

```
import org.apache.spark.mllib.tree.RandomForest
import org.apache.spark.mllib.tree.model.RandomForestModel
import org.apache.spark.mllib.util.MLUtils
// Load and parse the data file.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_
data.txt")
// Split the data into training and test sets (30% held out for
testing)
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
// Train a RandomForest model.
// 空的类别特征信息表示所有的特征都是连续的
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val numTrees = 3 // Use more in practice.
val featureSubsetStrategy = "auto" // Let the algorithm choose.
val impurity = "variance"
val maxDepth = 4
val maxBins = 32
val model = RandomForest.trainRegressor(trainingData, categorica
lFeaturesInfo,
  numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins)
// Evaluate model on test instances and compute test error
val labelsAndPredictions = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testMSE = labelsAndPredictions.map{ case(v, p) => math.pow((
v - p), 2)}.mean()
println("Test Mean Squared Error = " + testMSE)
println("Learned regression forest model:\n" + model.toDebugStri
ng)
```

5 源码分析

5.1 训练分析

训练过程简单可以分为两步，第一步是初始化，第二步是迭代构建随机森林。这两大步还分为若干小步，下面会分别介绍这些内容。

5.1.1 初始化

```
val retaggedInput = input.retag(classOf[LabeledPoint])
//建立决策树的元数据信息（分裂点位置、箱子数及各箱子包含特征属性的值等）
val metadata =
    DecisionTreeMetadata.buildMetadata(retaggedInput, strategy,
numTrees, featureSubsetStrategy)
//找到切分点（splits）及箱子信息（Bins）
//对于连续型特征，利用切分点抽样统计简化计算
//对于离散型特征，如果是无序的，则最多有个 splits=2^(numBins-1)-1 划分
//如果是有序的，则最多有 splits=numBins-1 个划分
val (splits, bins) = DecisionTree.findSplitsBins(retaggedInput,
metadata)
//转换成树形的 RDD 类型，转换后，所有样本点已经按分裂点条件分到了各自的箱子中

val treeInput = TreePoint.convertToTreeRDD(retaggedInput, bins,
metadata)
val withReplacement = if (numTrees > 1) true else false
// convertToBaggedRDD 方法使得每棵树就是样本的一个子集
val baggedInput = BaggedPoint.convertToBaggedRDD(treeInput,
          strategy.subsamplingRate, numTrees,
          withReplacement, seed).persist(StorageLevel.MEMORY_AND
_DISK)
//决策树的深度，最大为30
val maxDepth = strategy.maxDepth
//聚合的最大内存
val maxMemoryUsage: Long = strategy.maxMemoryInMB * 1024L * 1024
L
val maxMemoryPerNode = {
    val featureSubset: Option[Array[Int]] = if (metadata.subsamp
lingFeatures) {
        // Find numFeaturesPerNode largest bins to get an upper
bound on memory usage.
        Some(metadata.numBins.zipWithIndex.sortBy(- _._1)
          .take(metadata.numFeaturesPerNode).map(_._2))
    } else {
```

```
        None
    }
    //计算聚合操作时节点的内存
    RandomForest.aggregateSizeForNode(metadata, featureSubset) *
8L
}
```

初始化的第一步就是决策树元数据信息的构建。它的代码如下所示。

```
def buildMetadata(
      input: RDD[LabeledPoint],
      strategy: Strategy,
      numTrees: Int,
      featureSubsetStrategy: String): DecisionTreeMetadata = {
    //特征数
    val numFeatures = input.map(_.features.size).take(1).headOpt
ion.getOrElse {
      throw new IllegalArgumentException(s"DecisionTree requires
 size of input RDD > 0, " +
        s"but was given by empty one.")
    }
    val numExamples = input.count()
    val numClasses = strategy.algo match {
      case Classification => strategy.numClasses
      case Regression => 0
    }
    //最大可能的装箱数
    val maxPossibleBins = math.min(strategy.maxBins, numExamples
).toInt
    if (maxPossibleBins < strategy.maxBins) {
      logWarning(s"DecisionTree reducing maxBins from ${strategy
.maxBins} to $maxPossibleBins" +
        s" (= number of training instances)")
    }
    // We check the number of bins here against maxPossibleBins.
    // This needs to be checked here instead of in Strategy sinc
e maxPossibleBins can be modified
    // based on the number of training examples.
    //最大分类数要小于最大可能装箱数
```

```
    //这里categoricalFeaturesInfo是传入的信息，这个map保存特征的类别信
息。
    //例如，(n->k)表示特征k包含的类别有（0,1,...,k-1）
    if (strategy.categoricalFeaturesInfo.nonEmpty) {
      val maxCategoriesPerFeature = strategy.categoricalFeatures
Info.values.max
      val maxCategory =
        strategy.categoricalFeaturesInfo.find(_._2 == maxCategor
iesPerFeature).get._1
      require(maxCategoriesPerFeature <= maxPossibleBins,
        s"DecisionTree requires maxBins (= $maxPossibleBins) to
be at least as large as the " +
        s"number of values in each categorical feature, but cate
gorical feature $maxCategory " +
        s"has $maxCategoriesPerFeature values. Considering remov
e this and other categorical " +
        "features with a large number of values, or add more tra
ining examples.")
    }
    val unorderedFeatures = new mutable.HashSet[Int]()
    val numBins = Array.fill[Int](numFeatures)(maxPossibleBins)
    if (numClasses > 2) {
      // 多分类
      val maxCategoriesForUnorderedFeature =
        ((math.log(maxPossibleBins / 2 + 1) / math.log(2.0)) + 1
).floor.toInt
      strategy.categoricalFeaturesInfo.foreach { case (featureIn
dex, numCategories) =>
        //如果类别特征只有1个类，我们把它看成连续的特征
        if (numCategories > 1) {
          // Decide if some categorical features should be treat
ed as unordered features,
          //  which require 2 * ((1 << numCategories - 1) - 1) b
ins.
          // We do this check with log values to prevent overflo
ws in case numCategories is large.
          // The next check is equivalent to: 2 * ((1 << numCate
gories - 1) - 1) <= maxBins
          if (numCategories <= maxCategoriesForUnorderedFeature)
 {
```

```
            unorderedFeatures.add(featureIndex)
            numBins(featureIndex) = numUnorderedBins(numCategori
es)
          } else {
            numBins(featureIndex) = numCategories
          }
        }
      }
    } else {
      // 二分类或者回归
      strategy.categoricalFeaturesInfo.foreach { case (featureIn
dex, numCategories) =>
        //如果类别特征只有1个类，我们把它看成连续的特征
        if (numCategories > 1) {
          numBins(featureIndex) = numCategories
        }
      }
    }
    // 设置每个节点的特征数 (对随机森林而言).
    val _featureSubsetStrategy = featureSubsetStrategy match {
      case "auto" =>
        if (numTrees == 1) {//决策树时，使用所有特征
          "all"
        } else {
          if (strategy.algo == Classification) {//分类时，使用开平方

            "sqrt"
          } else { //回归时，使用1/3的特征
            "onethird"
          }
        }
      case _ => featureSubsetStrategy
    }
    val numFeaturesPerNode: Int = _featureSubsetStrategy match {
      case "all" => numFeatures
      case "sqrt" => math.sqrt(numFeatures).ceil.toInt
      case "log2" => math.max(1, (math.log(numFeatures) / math.l
og(2)).ceil.toInt)
      case "onethird" => (numFeatures / 3.0).ceil.toInt
    }
```

```
    new DecisionTreeMetadata(numFeatures, numExamples, numClasses, numBins.max,
      strategy.categoricalFeaturesInfo, unorderedFeatures.toSet, numBins,
      strategy.impurity, strategy.quantileCalculationStrategy, strategy.maxDepth,
      strategy.minInstancesPerNode, strategy.minInfoGain, numTrees, numFeaturesPerNode)
  }
```

初始化的第二步就是找到切分点（ `splits` ）及箱子信息（ `Bins` ）。这时，调用了 `DecisionTree.findSplitsBins` 方法，进入该方法了解详细信息。

```
/**
   * Returns splits and bins for decision tree calculation.
   * Continuous and categorical features are handled differently.
   *
   * Continuous features:
   *   For each feature, there are numBins - 1 possible splits representing the possible binary
   *   decisions at each node in the tree.
   *   This finds locations (feature values) for splits using a subsample of the data.
   *
   * Categorical features:
   *   For each feature, there is 1 bin per split.
   *   Splits and bins are handled in 2 ways:
   *   (a) "unordered features"
   *       For multiclass classification with a low-arity feature
   *       (i.e., if isMulticlass && isSpaceSufficientForAllCategoricalSplits),
   *       the feature is split based on subsets of categories.
   *   (b) "ordered features"
   *       For regression and binary classification,
   *       and for multiclass classification with a high-arity feature,
   *       there is one bin per category.
```

```
   *
   * @param input Training data: RDD of [[org.apache.spark.mllib
.regression.LabeledPoint]]
   * @param metadata Learning and dataset metadata
   * @return A tuple of (splits, bins).
   *         Splits is an Array of [[org.apache.spark.mllib.tree
.model.Split]]
   *          of size (numFeatures, numSplits).
   *         Bins is an Array of [[org.apache.spark.mllib.tree.m
odel.Bin]]
   *          of size (numFeatures, numBins).
   */
  protected[tree] def findSplitsBins(
      input: RDD[LabeledPoint],
      metadata: DecisionTreeMetadata): (Array[Array[Split]], Arr
ay[Array[Bin]]) = {
    //特征数
    val numFeatures = metadata.numFeatures
    // Sample the input only if there are continuous features.
    // 判断特征中是否存在连续特征
    val continuousFeatures = Range(0, numFeatures).filter(metada
ta.isContinuous)
    val sampledInput = if (continuousFeatures.nonEmpty) {
      // Calculate the number of samples for approximate quantil
e calculation.
      //采样样本数量，最少有 10000 个
      val requiredSamples = math.max(metadata.maxBins * metadata
.maxBins, 10000)
      //计算采样比例
      val fraction = if (requiredSamples < metadata.numExamples)
 {
        requiredSamples.toDouble / metadata.numExamples
      } else {
        1.0
      }
      //采样数据，有放回采样
      input.sample(withReplacement = false, fraction, new XORShi
ftRandom().nextInt())
    } else {
      input.sparkContext.emptyRDD[LabeledPoint]
```

```
    }
    //分裂点策略，目前 Spark 中只实现了一种策略：排序 Sort
    metadata.quantileStrategy match {
      case Sort =>
        findSplitsBinsBySorting(sampledInput, metadata, continuo
usFeatures)
      case MinMax =>
        throw new UnsupportedOperationException("minmax not supp
orted yet.")
      case ApproxHist =>
        throw new UnsupportedOperationException("approximate his
togram not supported yet.")
    }
  }
```

我们进入 `findSplitsBinsBySorting` 方法了解 `Sort` 分裂策略的实现。

```
private def findSplitsBinsBySorting(
      input: RDD[LabeledPoint],
      metadata: DecisionTreeMetadata,
      continuousFeatures: IndexedSeq[Int]): (Array[Array[Split]]
, Array[Array[Bin]]) = {
    def findSplits(
        featureIndex: Int,
        featureSamples: Iterable[Double]): (Int, (Array[Split],
Array[Bin])) = {
      //每个特征分别对应一组切分点位置，这里splits是有序的
      val splits = {
        // findSplitsForContinuousFeature 返回连续特征的所有切分位置
        val featureSplits = findSplitsForContinuousFeature(
          featureSamples.toArray,
          metadata,
          featureIndex)
        featureSplits.map(threshold => new Split(featureIndex, t
hreshold, Continuous, Nil))
      }
      //存放切分点位置对应的箱子信息
      val bins = {
        //采用最小阈值 Double.MinValue 作为最左边的分裂位置并进行装箱
```

```
        val lowSplit = new DummyLowSplit(featureIndex, Continuous)
        //最后一个箱子的计算采用最大阈值 Double.MaxValue 作为最右边的切分位置
        val highSplit = new DummyHighSplit(featureIndex, Continuous)
        // tack the dummy splits on either side of the computed splits
        val allSplits = lowSplit +: splits.toSeq :+ highSplit
        //将切分点两两结合成一个箱子
        allSplits.sliding(2).map {
          case Seq(left, right) => new Bin(left, right, Continuous, Double.MinValue)
        }.toArray
      }
      (featureIndex, (splits, bins))
    }
    val continuousSplits = {
      // reduce the parallelism for split computations when there are less
      // continuous features than input partitions. this prevents tasks from
      // being spun up that will definitely do no work.
      val numPartitions = math.min(continuousFeatures.length, input.partitions.length)
      input
        .flatMap(point => continuousFeatures.map(idx => (idx, point.features(idx))))
        .groupByKey(numPartitions)
        .map { case (k, v) => findSplits(k, v) }
        .collectAsMap()
    }
    val numFeatures = metadata.numFeatures
    //遍历所有特征
    val (splits, bins) = Range(0, numFeatures).unzip {
      //处理连续特征的情况
      case i if metadata.isContinuous(i) =>
        val (split, bin) = continuousSplits(i)
        metadata.setNumSplits(i, split.length)
        (split, bin)
```

```
        //处理离散特征且无序的情况
        case i if metadata.isCategorical(i) && metadata.isUnordere
d(i) =>
          // Unordered features
          // 2^(maxFeatureValue - 1) - 1 combinations
          val featureArity = metadata.featureArity(i)
          val split = Range(0, metadata.numSplits(i)).map { splitI
ndex =>
            val categories = extractMultiClassCategories(splitInde
x + 1, featureArity)
            new Split(i, Double.MinValue, Categorical, categories)
          }
          // For unordered categorical features, there is no need
to construct the bins.
          // since there is a one-to-one correspondence between th
e splits and the bins.
          (split.toArray, Array.empty[Bin])
        //处理离散特征且有序的情况
        case i if metadata.isCategorical(i) =>
          //有序特征无需处理，箱子与特征值对应
          // Ordered features
          // Bins correspond to feature values, so we do not need
to compute splits or bins
          // beforehand.  Splits are constructed as needed during
training.
          (Array.empty[Split], Array.empty[Bin])
      }
      (splits.toArray, bins.toArray)
    }
```

计算连续特征的所有切分位置需要调用方法 `findSplitsForContinuousFeature` 方法。

```
private[tree] def findSplitsForContinuousFeature(
        featureSamples: Array[Double],
        metadata: DecisionTreeMetadata,
        featureIndex: Int): Array[Double] = {
      val splits = {
        //切分数是bin的数量减1，即m-1
```

```
      val numSplits = metadata.numSplits(featureIndex)
      // （特征，特征出现的次数）
      val valueCountMap = featureSamples.foldLeft(Map.empty[Doub
le, Int]) { (m, x) =>
        m + ((x, m.getOrElse(x, 0) + 1))
      }
      // 根据特征进行排序
      val valueCounts = valueCountMap.toSeq.sortBy(_._1).toArray
      // if possible splits is not enough or just enough, just r
eturn all possible splits
      val possibleSplits = valueCounts.length
      //如果特征数小于切分数，所有特征均作为切分点
      if (possibleSplits <= numSplits) {
        valueCounts.map(_._1)
      } else {
        // 切分点之间的步长
        val stride: Double = featureSamples.length.toDouble / (n
umSplits + 1)
        val splitsBuilder = Array.newBuilder[Double]
        var index = 1
        // currentCount: sum of counts of values that have been
visited
        //第一个特征的出现次数
        var currentCount = valueCounts(0)._2
        // targetCount: target value for `currentCount`.
        // If `currentCount` is closest value to `targetCount`,
        // then current value is a split threshold.
        // After finding a split threshold, `targetCount` is add
ed by stride.
        // 如果currentCount离targetCount最近，那么当前值是切分点
        var targetCount = stride
        while (index < valueCounts.length) {
          val previousCount = currentCount
          currentCount += valueCounts(index)._2
          val previousGap = math.abs(previousCount - targetCount
)
          val currentGap = math.abs(currentCount - targetCount)
          // If adding count of current value to currentCount
          // makes the gap between currentCount and targetCount
smaller,
```

```
          // previous value is a split threshold.
          if (previousGap < currentGap) {
            splitsBuilder += valueCounts(index - 1)._1
            targetCount += stride
          }
          index += 1
        }
        splitsBuilder.result()
      }
    }
    splits
  }
```

5.1.2 迭代构建随机森林

```
//节点是否使用缓存，节点 ID 从 1 开始，1 即为这颗树的根节点，左节点为 2，
右节点为 3，依次递增下去
val nodeIdCache = if (strategy.useNodeIdCache) {
   Some(NodeIdCache.init(
        data = baggedInput,
        numTrees = numTrees,
        checkpointInterval = strategy.checkpointInterval,
        initVal = 1))
} else {
   None
}
// FIFO queue of nodes to train: (treeIndex, node)
val nodeQueue = new mutable.Queue[(Int, Node)]()
val rng = new scala.util.Random()
rng.setSeed(seed)
// Allocate and queue root nodes.
//创建树的根节点
val topNodes: Array[Node] = Array.fill[Node](numTrees)(Node.empt
yNode(nodeIndex = 1))
//将（树的索引，树的根节点）入队，树索引从 0 开始，根节点从 1 开始
Range(0, numTrees).foreach(treeIndex => nodeQueue.enqueue((treeI
ndex, topNodes(treeIndex))))
while (nodeQueue.nonEmpty) {
    // Collect some nodes to split, and choose features for each
 node (if subsampling).
    // Each group of nodes may come from one or multiple trees,
and at multiple levels.
    // 取得每个树所有需要切分的节点,nodesForGroup表示需要切分的节点
    val (nodesForGroup, treeToNodeToIndexInfo) =
        RandomForest.selectNodesToSplit(nodeQueue, maxMemoryUsag
e, metadata, rng)
    //找出最优切点
    DecisionTree.findBestSplits(baggedInput, metadata, topNodes,
 nodesForGroup,
        treeToNodeToIndexInfo, splits, bins, nodeQueue, timer, n
odeIdCache = nodeIdCache)
}
```

这里有两点需要重点介绍，第一点是取得每个树所有需要切分的节点，通过 `RandomForest.selectNodesToSplit` 方法实现；第二点是找出最优的切分，通过 `DecisionTree.findBestSplits` 方法实现。下面分别介绍这两点。

- 取得每个树所有需要切分的节点

```
 private[tree] def selectNodesToSplit(
      nodeQueue: mutable.Queue[(Int, Node)],
      maxMemoryUsage: Long,
      metadata: DecisionTreeMetadata,
      rng: scala.util.Random): (Map[Int, Array[Node]], Map[Int,
Map[Int, NodeIndexInfo]]) = {
    // nodesForGroup保存需要切分的节点，treeIndex --> nodes
    val mutableNodesForGroup = new mutable.HashMap[Int, mutable.
ArrayBuffer[Node]]()
    // mutableTreeToNodeToIndexInfo保存每个节点中选中特征的索引
    // treeIndex --> (global) node index --> (node index in grou
p, feature indices)
    //(global) node index是树中的索引，组中节点索引的范围是[0, numNode
sInGroup)
    val mutableTreeToNodeToIndexInfo =
      new mutable.HashMap[Int, mutable.HashMap[Int, NodeIndexInfo
]]()
    var memUsage: Long = 0L
    var numNodesInGroup = 0
    while (nodeQueue.nonEmpty && memUsage < maxMemoryUsage) {
      val (treeIndex, node) = nodeQueue.head
      // Choose subset of features for node (if subsampling).
      // 选中特征子集
      val featureSubset: Option[Array[Int]] = if (metadata.subsa
mplingFeatures) {
        Some(SamplingUtils.reservoirSampleAndCount(Range(0,
          metadata.numFeatures).iterator, metadata.numFeaturesPe
rNode, rng.nextLong)._1)
      } else {
        None
      }
      // Check if enough memory remains to add this node to the
group.
      // 检查是否有足够的内存
```

```
      val nodeMemUsage = RandomForest.aggregateSizeForNode(metad
ata, featureSubset) * 8L
      if (memUsage + nodeMemUsage <= maxMemoryUsage) {
        nodeQueue.dequeue()
        mutableNodesForGroup.getOrElseUpdate(treeIndex, new muta
ble.ArrayBuffer[Node]()) += node
        mutableTreeToNodeToIndexInfo
          .getOrElseUpdate(treeIndex, new mutable.HashMap[Int, N
odeIndexInfo]())(node.id)
          = new NodeIndexInfo(numNodesInGroup, featureSubset)
      }
      numNodesInGroup += 1
      memUsage += nodeMemUsage
    }
    // 将可变map转换为不可变map
    val nodesForGroup: Map[Int, Array[Node]] = mutableNodesForGr
oup.mapValues(_.toArray).toMap
    val treeToNodeToIndexInfo = mutableTreeToNodeToIndexInfo.map
Values(_.toMap).toMap
    (nodesForGroup, treeToNodeToIndexInfo)
  }
```

- 选中最优切分

```
//所有可切分的节点
val nodes = new Array[Node](numNodes)
nodesForGroup.foreach { case (treeIndex, nodesForTree) =>
   nodesForTree.foreach { node =>
     nodes(treeToNodeToIndexInfo(treeIndex)(node.id).nodeIndexIn
Group) = node
   }
}
// In each partition, iterate all instances and compute aggregat
e stats for each node,
// yield an (nodeIndex, nodeAggregateStats) pair for each node.
// After a `reduceByKey` operation,
// stats of a node will be shuffled to a particular partition an
d be combined together,
// then best splits for nodes are found there.
```

```
// Finally, only best Splits for nodes are collected to driver to construct decision tree.
//获取节点对应的特征
val nodeToFeatures = getNodeToFeatures(treeToNodeToIndexInfo)
val nodeToFeaturesBc = input.sparkContext.broadcast(nodeToFeatures)
val partitionAggregates : RDD[(Int, DTStatsAggregator)] = if (nodeIdCache.nonEmpty) {
    input.zip(nodeIdCache.get.nodeIdsForInstances).mapPartitions { points =>
      // Construct a nodeStatsAggregators array to hold node aggregate stats,
      // each node will have a nodeStatsAggregator
      val nodeStatsAggregators = Array.tabulate(numNodes) { nodeIndex =>
          //节点对应的特征集
          val featuresForNode = nodeToFeaturesBc.value.flatMap { nodeToFeatures =>
            Some(nodeToFeatures(nodeIndex))
          }
          // DTStatsAggregator，其中引用了 ImpurityAggregator，给出计算不纯度 impurity 的逻辑
          new DTStatsAggregator(metadata, featuresForNode)
      }
      // 迭代当前分区的所有对象，更新聚合统计信息，统计信息即采样数据的权重值

      points.foreach(binSeqOpWithNodeIdCache(nodeStatsAggregators, _))
      // transform nodeStatsAggregators array to (nodeIndex, nodeAggregateStats) pairs,
      // which can be combined with other partition using `reduceByKey`
      nodeStatsAggregators.view.zipWithIndex.map(_.swap).iterator
    }
} else {
      input.mapPartitions { points =>
        // Construct a nodeStatsAggregators array to hold node aggregate stats,
        // each node will have a nodeStatsAggregator
```

```
        val nodeStatsAggregators = Array.tabulate(numNodes) { no
deIndex =>
          //节点对应的特征集
          val featuresForNode = nodeToFeaturesBc.value.flatMap {
 nodeToFeatures =>
            Some(nodeToFeatures(nodeIndex))
          }
          // DTStatsAggregator，其中引用了 ImpurityAggregator，给出
计算不纯度 impurity 的逻辑
          new DTStatsAggregator(metadata, featuresForNode)
        }
        // 迭代当前分区的所有对象，更新聚合统计信息
        points.foreach(binSeqOp(nodeStatsAggregators, _))
        // transform nodeStatsAggregators array to (nodeIndex, n
odeAggregateStats) pairs,
        // which can be combined with other partition using `red
uceByKey`
        nodeStatsAggregators.view.zipWithIndex.map(_.swap).itera
tor
      }
}
val nodeToBestSplits = partitionAggregates.reduceByKey((a, b) =>
 a.merge(b))
    .map { case (nodeIndex, aggStats) =>
          val featuresForNode = nodeToFeaturesBc.value.map { nod
eToFeatures =>
            nodeToFeatures(nodeIndex)
    }
    // find best split for each node
    val (split: Split, stats: InformationGainStats, predict: Pre
dict) =
        binsToBestSplit(aggStats, splits, featuresForNode, nodes
(nodeIndex))
    (nodeIndex, (split, stats, predict))
}.collectAsMap()
```

该方法中的关键是对`binsToBestSplit`方法的调用，`binsToBestSplit`方法代码如下：

```
private def binsToBestSplit(
      binAggregates: DTStatsAggregator,
      splits: Array[Array[Split]],
      featuresForNode: Option[Array[Int]],
      node: Node): (Split, InformationGainStats, Predict) = {
    // 如果当前节点是根节点，计算预测和不纯度
    val level = Node.indexToLevel(node.id)
    var predictWithImpurity: Option[(Predict, Double)] = if (lev
el == 0) {
      None
    } else {
      Some((node.predict, node.impurity))
    }
    // 对各特征及切分点，计算其信息增益并从中选择最优 (feature, split)
    val (bestSplit, bestSplitStats) =
      Range(0, binAggregates.metadata.numFeaturesPerNode).map {
featureIndexIdx =>
      val featureIndex = if (featuresForNode.nonEmpty) {
        featuresForNode.get.apply(featureIndexIdx)
      } else {
        featureIndexIdx
      }
      val numSplits = binAggregates.metadata.numSplits(featureIn
dex)
       //特征为连续值的情况
      if (binAggregates.metadata.isContinuous(featureIndex)) {
        // Cumulative sum (scanLeft) of bin statistics.
        // Afterwards, binAggregates for a bin is the sum of agg
regates for
        // that bin + all preceding bins.
        val nodeFeatureOffset = binAggregates.getFeatureOffset(f
eatureIndexIdx)
        var splitIndex = 0
        while (splitIndex < numSplits) {
          binAggregates.mergeForFeature(nodeFeatureOffset, split
Index + 1, splitIndex)
          splitIndex += 1
        }
        // Find best split.
```

```
        val (bestFeatureSplitIndex, bestFeatureGainStats) =
          Range(0, numSplits).map { case splitIdx =>
            //计算 leftChild 及 rightChild 子节点的 impurity
            val leftChildStats = binAggregates.getImpurityCalcul
ator(nodeFeatureOffset, splitIdx)
            val rightChildStats = binAggregates.getImpurityCalcu
lator(nodeFeatureOffset, numSplits)
            rightChildStats.subtract(leftChildStats)
            //求 impurity 的预测值，采用的是平均值计算
            predictWithImpurity = Some(predictWithImpurity.getOr
Else(
              calculatePredictImpurity(leftChildStats, rightChil
dStats)))
            //求信息增益 information gain 值，用于评估切分点是否最优,
请参考决策树中1.4.4章节的介绍
            val gainStats = calculateGainForSplit(leftChildStats
,
              rightChildStats, binAggregates.metadata, predictWi
thImpurity.get._2)
            (splitIdx, gainStats)
          }.maxBy(_._2.gain)
        (splits(featureIndex)(bestFeatureSplitIndex), bestFeatur
eGainStats)
      }
      //无序离散特征时的情况
      else if (binAggregates.metadata.isUnordered(featureIndex))
 {
        // Unordered categorical feature
        val (leftChildOffset, rightChildOffset) =
          binAggregates.getLeftRightFeatureOffsets(featureIndexI
dx)
        val (bestFeatureSplitIndex, bestFeatureGainStats) =
          Range(0, numSplits).map { splitIndex =>
            val leftChildStats = binAggregates.getImpurityCalcul
ator(leftChildOffset, splitIndex)
            val rightChildStats = binAggregates.getImpurityCalcu
lator(rightChildOffset, splitIndex)
            predictWithImpurity = Some(predictWithImpurity.getOr
Else(
              calculatePredictImpurity(leftChildStats, rightChil
```

```
dStats)))
            val gainStats = calculateGainForSplit(leftChildStats
,
              rightChildStats, binAggregates.metadata, predictWi
thImpurity.get._2)
            (splitIndex, gainStats)
          }.maxBy(_._2.gain)
        (splits(featureIndex)(bestFeatureSplitIndex), bestFeatur
eGainStats)
      } else {//有序离散特征时的情况
        // Ordered categorical feature
        val nodeFeatureOffset = binAggregates.getFeatureOffset(f
eatureIndexIdx)
        val numBins = binAggregates.metadata.numBins(featureInde
x)
        /* Each bin is one category (feature value).
         * The bins are ordered based on centroidForCategories,
and this ordering determines which
         * splits are considered.  (With K categories, we consid
er K - 1 possible splits.)
         *
         * centroidForCategories is a list: (category, centroid)
         */
        //多元分类时的情况
        val centroidForCategories = if (binAggregates.metadata.i
sMulticlass) {
          // For categorical variables in multiclass classificat
ion,
          // the bins are ordered by the impurity of their corre
sponding labels.
          Range(0, numBins).map { case featureValue =>
            val categoryStats = binAggregates.getImpurityCalcula
tor(nodeFeatureOffset, featureValue)
            val centroid = if (categoryStats.count != 0) {
              // impurity 求的就是均方差
              categoryStats.calculate()
            } else {
              Double.MaxValue
            }
            (featureValue, centroid)
```

```
          }
        } else { // 回归或二元分类时的情况
          // For categorical variables in regression and binary
classification,
          // the bins are ordered by the centroid of their corre
sponding labels.
          Range(0, numBins).map { case featureValue =>
            val categoryStats = binAggregates.getImpurityCalcula
tor(nodeFeatureOffset, featureValue)
            val centroid = if (categoryStats.count != 0) {
              //求的就是平均值作为 impurity
              categoryStats.predict
            } else {
              Double.MaxValue
            }
            (featureValue, centroid)
          }
        }
        // bins sorted by centroids
        val categoriesSortedByCentroid = centroidForCategories.t
oList.sortBy(_._2)
        // Cumulative sum (scanLeft) of bin statistics.
        // Afterwards, binAggregates for a bin is the sum of agg
regates for
        // that bin + all preceding bins.
        var splitIndex = 0
        while (splitIndex < numSplits) {
          val currentCategory = categoriesSortedByCentroid(split
Index)._1
          val nextCategory = categoriesSortedByCentroid(splitInd
ex + 1)._1
          //将两个箱子的状态信息进行合并
          binAggregates.mergeForFeature(nodeFeatureOffset, nextC
ategory, currentCategory)
          splitIndex += 1
        }
        // lastCategory = index of bin with total aggregates for
 this (node, feature)
        val lastCategory = categoriesSortedByCentroid.last._1
        // Find best split.
```

```
        //通过信息增益值选择最优切分点
        val (bestFeatureSplitIndex, bestFeatureGainStats) =
          Range(0, numSplits).map { splitIndex =>
            val featureValue = categoriesSortedByCentroid(splitI
ndex)._1
            val leftChildStats =
              binAggregates.getImpurityCalculator(nodeFeatureOff
set, featureValue)
            val rightChildStats =
              binAggregates.getImpurityCalculator(nodeFeatureOff
set, lastCategory)
            rightChildStats.subtract(leftChildStats)
            predictWithImpurity = Some(predictWithImpurity.getOr
Else(
              calculatePredictImpurity(leftChildStats, rightChil
dStats)))
            val gainStats = calculateGainForSplit(leftChildStats
,
              rightChildStats, binAggregates.metadata, predictWi
thImpurity.get._2)
            (splitIndex, gainStats)
          }.maxBy(_._2.gain)
        val categoriesForSplit =
          categoriesSortedByCentroid.map(_._1.toDouble).slice(0,
 bestFeatureSplitIndex + 1)
        val bestFeatureSplit =
          new Split(featureIndex, Double.MinValue, Categorical,
categoriesForSplit)
        (bestFeatureSplit, bestFeatureGainStats)
      }
    }.maxBy(_._2.gain)
    (bestSplit, bestSplitStats, predictWithImpurity.get._1)
  }
```

5.2 预测分析

在利用随机森林进行预测时，调用的 `predict` 方法扩展自 `TreeEnsembleModel` ，它是树结构组合模型的表示，其核心代码如下所示：

```
//不同的策略采用不同的预测方法
def predict(features: Vector): Double = {
    (algo, combiningStrategy) match {
      case (Regression, Sum) =>
        predictBySumming(features)
      case (Regression, Average) =>
        predictBySumming(features) / sumWeights
      case (Classification, Sum) => // binary classification
        val prediction = predictBySumming(features)
        // TODO: predicted labels are +1 or -1 for GBT. Need a b
etter way to store this info.
        if (prediction > 0.0) 1.0 else 0.0
      case (Classification, Vote) =>
        predictByVoting(features)
      case _ =>
        throw new IllegalArgumentException()
    }
  }
private def predictBySumming(features: Vector): Double = {
    val treePredictions = trees.map(_.predict(features))
    //两个向量的内集
    blas.ddot(numTrees, treePredictions, 1, treeWeights, 1)
}
//通过投票选举
private def predictByVoting(features: Vector): Double = {
    val votes = mutable.Map.empty[Int, Double]
    trees.view.zip(treeWeights).foreach { case (tree, weight) =>
      val prediction = tree.predict(features).toInt
      votes(prediction) = votes.getOrElse(prediction, 0.0) + wei
ght
    }
    votes.maxBy(_._2)._1
}
```

参考文献

【1】机器学习.周志华

【2】Spark 随机森林算法原理、源码分析及案例实战

【3】Scalable Distributed Decision Trees in Spark MLlib

梯度提升树

1 Boosting

`Boosting` 是一类将弱学习器提升为强学习器的算法。这类算法的工作机制类似：先从初始训练集中训练出一个基学习器，再根据基学习器的表现对训练样本分布进行调整，使得先前基学习器做错的训练样本在后续受到更多关注。然后基于调整后的样本分布来训练下一个基学习器；如此重复进行，直至基学习器的数目达到事先指定的值 `T` ，最终将这 `T` 个基学习器进行加权结合。

`Boost` 算法是在算法开始时，为每一个样本赋上一个相等的权重值，也就是说，最开始的时候，大家都是一样重要的。在每一次训练中得到的模型，会使得数据点的估计有所差异，所以在每一步结束后，我们需要对权重值进行处理，而处理的方式就是通过增加错分点的权重，这样使得某些点如果老是被分错，那么就会被“严重关注”，也就被赋上一个很高的权重。然后等进行了 `N` 次迭代，将会得到 `N` 个简单的基分类器（ `basic learner` ），最后将它们组合起来，可以对它们进行加权（错误率越大的基分类器权重值越小，错误率越小的基分类器权重值越大）、或者让它们进行投票等得到一个最终的模型。

梯度提升（ `gradient boosting` ）属于 `Boost` 算法的一种，也可以说是 `Boost` 算法的一种改进，它与传统的 `Boost` 有着很大的区别，它的每一次计算都是为了减少上一次的残差(`residual`)，而为了减少这些残差，可以在残差减少的梯度(`Gradient`)方向上建立一个新模型。所以说，在 `Gradient Boost` 中，每个新模型的建立是为了使得先前模型残差往梯度方向减少，与传统的 `Boost` 算法对正确、错误的样本进行加权有着极大的区别。

梯度提升算法的核心在于，每棵树是从先前所有树的残差中来学习。利用的是当前模型中损失函数的负梯度值作为提升树算法中的残差的近似值，进而拟合一棵回归（分类）树。

2 梯度提升

根据参考文献【1】的介绍，梯度提升算法的算法流程如下所示：

	Algorithm 1: Gradient_TreeBoost
1	$F_0(\mathbf{x}) = \arg\min_\gamma \sum_{i=1}^N \Psi(y_i, \gamma)$
2	For $m = 1$ to M do:
3	$\tilde{y}_{im} = -\left[\frac{\partial\Psi(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}, \ i = 1, N$
4	$\{R_{lm}\}_1^L = L - \text{terminal node } tree(\{\tilde{y}_{im}, \mathbf{x}_i\}_1^N)$
5	$\gamma_{lm} = \arg\min_\gamma \sum_{\mathbf{x}_i \in R_{lm}} \Psi(y_i, F_{m-1}(\mathbf{x}_i) + \gamma)$
6	$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \nu \cdot \gamma_{lm} 1(\mathbf{x} \in R_{lm})$
7	endFor

在上述的流程中，`F(x)` 表示学习器，`psi` 表示损失函数，第3行的 `y_im` 表示负梯度方向，第4行的 `R_lm` 表示原数据改变分布后的数据。

在 `MLlib` 中，提供的损失函数有三种。如下图所示。

Loss	Task	Formula	Description
Log Loss	Classification	$2\sum_{i=1}^N \log(1+\exp(-2y_i F(x_i)))$	Twice binomial negative log likelihood.
Squared Error	Regression	$\sum_{i=1}^N (y_i - F(x_i))^2$	Also called L2 loss. Default loss for regression tasks.
Absolute Error	Regression	$\sum_{i=1}^N \lvert y_i - F(x_i)\rvert$	Also called L1 loss. Can be more robust to outliers than Squared Error.

第一个对数损失用于分类，后两个平方误差和绝对误差用于回归。

3 随机梯度提升

有文献证明，注入随机性到上述的过程中可以提高函数估计的性能。受到 `Breiman` 的影响，将随机性作为一个考虑的因素。在每次迭代中，随机的在训练集中抽取一个子样本集，然后在后续的操作中用这个子样本集代替全体样本。 这就形成了随机梯度提升算法。它的流程如下所示：

	Algorithm 2: Stochastic Gradient_TreeBoost
1	$F_0(\mathbf{x}) = \arg\min_{\gamma} \sum_{i=1}^{N} \Psi\left(y_i, \gamma\right)$
2	For $m = 1$ to M do:
3	$\{\pi(i)\}_1^N = \text{rand_perm}\ \{i\}_1^N$
4	$\tilde{y}_{\pi(i)m} = -\left[\frac{\partial\Psi(y_{\pi(i)}, F(\mathbf{x}_{\pi(i)}))}{\partial F(\mathbf{x}_{\pi(i)})}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})},\ i = 1, \tilde{N}$
5	$\{R_{lm}\}_1^L = L - \text{terminal node}\ tree(\{\tilde{y}_{\pi(i)m}, \mathbf{x}_{\pi(i)}\}_1^{\tilde{N}})$
6	$\gamma_{lm} = \arg\min_{\gamma} \sum_{\mathbf{x}_{\pi(i)} \in R_{lm}} \Psi\left(y_{\pi(i)}, F_{m-1}(\mathbf{x}_{\pi(i)}) + \gamma\right)$
7	$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \nu \cdot \gamma_{lm} 1(\mathbf{x} \in R_{lm})$
8	endFor

4 实例

下面的代码是分类的例子。

```
import org.apache.spark.mllib.tree.GradientBoostedTrees
import org.apache.spark.mllib.tree.configuration.BoostingStrategy

import org.apache.spark.mllib.tree.model.GradientBoostedTreesMod
el
import org.apache.spark.mllib.util.MLUtils
// 准备数据
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_
data.txt")
// Split the data into training and test sets (30% held out for
testing)
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
// 训练模型
// The defaultParams for Classification use LogLoss by default.
val boostingStrategy = BoostingStrategy.defaultParams("Classific
ation")
boostingStrategy.numIterations = 3 // Note: Use more iterations
in practice.
boostingStrategy.treeStrategy.numClasses = 2
boostingStrategy.treeStrategy.maxDepth = 5
// Empty categoricalFeaturesInfo indicates all features are cont
inuous.
boostingStrategy.treeStrategy.categoricalFeaturesInfo = Map[Int,
Int]()
val model = GradientBoostedTrees.train(trainingData, boostingStr
ategy)
// 用测试数据评价模型
val labelAndPreds = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDo
uble / testData.count()
println("Test Error = " + testErr)
println("Learned classification GBT model:\n" + model.toDebugStr
ing)
```

下面的代码是回归的例子。

```
import org.apache.spark.mllib.tree.GradientBoostedTrees
import org.apache.spark.mllib.tree.configuration.BoostingStrategy

import org.apache.spark.mllib.tree.model.GradientBoostedTreesMod
el
import org.apache.spark.mllib.util.MLUtils
// 准备数据
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_
data.txt")
// Split the data into training and test sets (30% held out for
testing)
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
// 训练模型
// The defaultParams for Regression use SquaredError by default.
val boostingStrategy = BoostingStrategy.defaultParams("Regressio
n")
boostingStrategy.numIterations = 3 // Note: Use more iterations
in practice.
boostingStrategy.treeStrategy.maxDepth = 5
// Empty categoricalFeaturesInfo indicates all features are cont
inuous.
boostingStrategy.treeStrategy.categoricalFeaturesInfo = Map[Int,
Int]()
val model = GradientBoostedTrees.train(trainingData, boostingStr
ategy)
// 用测试数据评价模型
val labelsAndPredictions = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testMSE = labelsAndPredictions.map{ case(v, p) => math.pow((
v - p), 2)}.mean()
println("Test Mean Squared Error = " + testMSE)
println("Learned regression GBT model:\n" + model.toDebugString)
```

5 源码分析

5.1 训练分析

梯度提升树的训练从 `run` 方法开始。

```
def run(input: RDD[LabeledPoint]): GradientBoostedTreesModel = {
    val algo = boostingStrategy.treeStrategy.algo
    algo match {
      case Regression =>
        GradientBoostedTrees.boost(input, input, boostingStrateg
y, validate = false)
      case Classification =>
        // Map labels to -1, +1 so binary classification can be
treated as regression.
        val remappedInput = input.map(x => new LabeledPoint((x.l
abel * 2) - 1, x.features))
        GradientBoostedTrees.boost(remappedInput, remappedInput,
 boostingStrategy, validate = false)
      case _ =>
        throw new IllegalArgumentException(s"$algo is not suppor
ted by the gradient boosting.")
    }
}
```

在 `MLlib` 中，梯度提升树只能用于二分类和回归。所以，在上面的代码中，将标签映射为 `-1,+1` ，那么二分类也可以被当做回归。整个训练过程在 `GradientBoostedTrees.boost` 中实现。 `GradientBoostedTrees.boost` 的过程分为三步，第一步，初始化参数；第二步，训练第一棵树；第三步，迭代训练后续的树。下面分别介绍这三步。

- 初始化参数

```
// 初始化梯度提升参数
// 迭代次数，默认为100
val numIterations = boostingStrategy.numIterations
// 基学习器
val baseLearners = new Array[DecisionTreeModel](numIterations)
// 基学习器权重
val baseLearnerWeights = new Array[Double](numIterations)
// 损失函数，分类时，用对数损失，回归时，用误差平方损失
val loss = boostingStrategy.loss
val learningRate = boostingStrategy.learningRate
// Prepare strategy for individual trees, which use regression w
ith variance impurity.
// 回归时，使用方差计算不纯度
val treeStrategy = boostingStrategy.treeStrategy.copy
val validationTol = boostingStrategy.validationTol
treeStrategy.algo = Regression
treeStrategy.impurity = Variance
// 缓存输入数据
val persistedInput = if (input.getStorageLevel == StorageLevel.N
ONE) {
   input.persist(StorageLevel.MEMORY_AND_DISK)
   true
} else {
   false
}
// Prepare periodic checkpointers
val predErrorCheckpointer = new PeriodicRDDCheckpointer[(Double,
Double)](
    treeStrategy.getCheckpointInterval, input.sparkContext)
val validatePredErrorCheckpointer = new PeriodicRDDCheckpointer[(
Double, Double)](
    treeStrategy.getCheckpointInterval, input.sparkContext)
```

- 训练第一棵树（即第一个基学习器）

```
//通过训练数据训练出一颗决策树，具体信息请参考随机森林的分析
val firstTreeModel = new DecisionTree(treeStrategy).run(input)
val firstTreeWeight = 1.0
baseLearners(0) = firstTreeModel
baseLearnerWeights(0) = firstTreeWeight
var predError: RDD[(Double, Double)] = GradientBoostedTreesModel
.
      computeInitialPredictionAndError(input, firstTreeWeight, f
irstTreeModel, loss)
predErrorCheckpointer.update(predError)
```

这里比较关键的是通过 `GradientBoostedTreesModel.computeInitialPredictionAndError` 计算初始的预测和误差。

```
def computeInitialPredictionAndError(
      data: RDD[LabeledPoint],
      initTreeWeight: Double,
      initTree: DecisionTreeModel,
      loss: Loss): RDD[(Double, Double)] = {
    data.map { lp =>
      val pred = initTreeWeight * initTree.predict(lp.features)
      val error = loss.computeError(pred, lp.label)
      (pred, error)
    }
  }
```

根据选择的损失函数的不同，`computeError` 的实现不同。

```
//对数损失的实现
override private[mllib] def computeError(prediction: Double, label: Double): Double = {
    val margin = 2.0 * label * prediction
    // The following is equivalent to 2.0 * log(1 + exp(-margin)) but more numerically stable.
    2.0 * MLUtils.log1pExp(-margin)
}
//误差平方损失
override private[mllib] def computeError(prediction: Double, label: Double): Double = {
    val err = label - prediction
    err * err
}
```

- 迭代训练后续树

```
var validatePredError: RDD[(Double, Double)] = GradientBoostedTreesModel.
      computeInitialPredictionAndError(validationInput, firstTreeWeight, firstTreeModel, loss)
if (validate) validatePredErrorCheckpointer.update(validatePredError)
var bestValidateError = if (validate) validatePredError.values.mean() else 0.0
var bestM = 1
var m = 1
var doneLearning = false
while (m < numIterations && !doneLearning) {
    // Update data with pseudo-residuals
    // 根据梯度调整训练数据
    val data = predError.zip(input).map { case ((pred, _), point) =>
      //标签为上一棵树预测的数据的负梯度方向
      LabeledPoint(-loss.gradient(pred, point.label), point.features)
    }
    //训练下一棵树
    val model = new DecisionTree(treeStrategy).run(data)
```

```
    // Update partial model
    baseLearners(m) = model
    // Note: The setting of baseLearnerWeights is incorrect for
losses other than SquaredError.
    //        Technically, the weight should be optimized for the
 particular loss.
    //        However, the behavior should be reasonable, though
not optimal.
    baseLearnerWeights(m) = learningRate
    //更新预测和误差
    predError = GradientBoostedTreesModel.updatePredictionError(
        input, predError, baseLearnerWeights(m), baseLearners(m)
, loss)
    predErrorCheckpointer.update(predError)
    //当需要验证阈值，提前终止迭代时
    if (validate) {
        // Stop training early if
        // 1. Reduction in error is less than the validationTol
or
        // 2. If the error increases, that is if the model is ov
erfit.
        // We want the model returned corresponding to the best
validation error.
        validatePredError = GradientBoostedTreesModel.updatePred
ictionError(
          validationInput, validatePredError, baseLearnerWeights
(m), baseLearners(m), loss)
        validatePredErrorCheckpointer.update(validatePredError)
        val currentValidateError = validatePredError.values.mean
()
        if (bestValidateError - currentValidateError < validatio
nTol * Math.max(
          currentValidateError, 0.01)) {
          doneLearning = true
        } else if (currentValidateError < bestValidateError) {
          bestValidateError = currentValidateError
          bestM = m + 1
        }
    }
    m += 1
```

```
}
```

上面代码最重要的部分是更新预测和误差的实现。通过 `GradientBoostedTreesModel.updatePredictionError` 实现。

```
def updatePredictionError(
    data: RDD[LabeledPoint],
    predictionAndError: RDD[(Double, Double)],
    treeWeight: Double,
    tree: DecisionTreeModel,
    loss: Loss): RDD[(Double, Double)] = {
    val newPredError = data.zip(predictionAndError).mapPartition
s { iter =>
      iter.map { case (lp, (pred, error)) =>
        val newPred = pred + tree.predict(lp.features) * treeWei
ght
        val newError = loss.computeError(newPred, lp.label)
        (newPred, newError)
      }
    }
    newPredError
  }
```

5.2 测试

利用梯度提升树进行预测时，调用的 `predict` 方法扩展自 `TreeEnsembleModel` ，它是树结构组合模型的表示，其核心代码如下所示：

```
//不同的策略采用不同的预测方法
def predict(features: Vector): Double = {
    (algo, combiningStrategy) match {
      case (Regression, Sum) =>
        predictBySumming(features)
      case (Regression, Average) =>
        predictBySumming(features) / sumWeights
      //用于梯度提升树，转换为1 或者 0
      case (Classification, Sum) => // binary classification
        val prediction = predictBySumming(features)
        // TODO: predicted labels are +1 or -1 for GBT. Need a b
etter way to store this info.
        if (prediction > 0.0) 1.0 else 0.0
      case (Classification, Vote) =>
        predictByVoting(features)
      case _ =>
        throw new IllegalArgumentException()
    }
}
private def predictBySumming(features: Vector): Double = {
    val treePredictions = trees.map(_.predict(features))
    //两个向量的内集
    blas.ddot(numTrees, treePredictions, 1, treeWeights, 1)
}
```

参考文献

【1】Stochastic Gradient Boost

【2】机器学习算法-梯度树提升GTB（GBRT）

保序回归

1 保序回归

保序回归解决了下面的问题：给定包含 `n` 个数据点的序列 `y_1,y_2,...,y_n` ，怎样通过一个单调的序列 `beta_1,beta_2,...,beta_n` 来归纳这个问题。形式上，这个问题就是为了找到

$$\hat{\beta}^{\text{iso}} = \underset{\beta \in \mathbb{R}^n}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - \beta_i)^2 \ \text{subject to} \ \beta_1 \leq \ldots \leq \beta_n. \tag{1}$$

大部分时候，我们会在括号前加上权重 `w_i` 。解决这个问题的一个方法就是 `pool adjacent violators algorithm(PAVA)` 算法。粗略的讲， `PAVA` 算法的过程描述如下：

我们从左边的 `y_1` 开始，右移 `y_1` 直到我们遇到第一个违例(`violation`)即 `y_i < y_i+1` ，然后，我们用违例之前的所有 `y` 的平方替换这些 `y` ，以满足单调性。我们继续这个过程，直到我们最后到达 `y_n` 。

2 近似保序

给定一个序列 `y_1,y_2,...,y_n` ，我们寻找一个近似单调估计，考虑下面的问题

$$\hat{\beta}_\lambda = \underset{\beta \in \mathbb{R}^n}{\operatorname{argmin}} \frac{1}{2} \sum_{i=1}^{n} (y_i - \beta_i)^2 + \lambda \sum_{i=1}^{n-1} (\beta_i - \beta_{i+1})_+, \tag{2}$$

在上式中， `X_+` 表示正数部分，即 `X_+ = X.1 (x>0)` 。这是一个凸优化问题，处罚项处罚违反单调性（即 `beta_i > beta_i+1` ）的邻近对。

在公式（2）中，隐含着一个假设，即使用等距的网格测量数据点。如果情况不是这样，那么可以修改惩罚项为下面的形式

$$\lambda \sum_{i=1}^{n-1} \frac{(\beta_i - \beta_{i+1})_+}{x_{i+1} - x_i},$$

`x_i` 表示 `y_i` 测量得到的值。

3 近似保序算法流程

这个算法是标准 `PAVA` 算法的修改版本，它并不从数据的左端开始，而是在需要时连接相邻的点，以产生近似保序最优的顺序。相比一下，`PAVA` 对中间的序列并不特别感兴趣，只关心最后的序列。

有下面一个引理成立。

Lemma 1. *Suppose that for some* λ, *we have two adjacent coordinates of the solution satisfying* $\hat{\beta}_{\lambda,i} = \hat{\beta}_{\lambda,i+1}$. *Then* $\hat{\beta}_{\lambda_0,i} = \hat{\beta}_{\lambda_0,i+1}$ *for all* $\lambda_0 > \lambda$.

这个引理证明的事实极大地简化了近似保序解路径（`solution path`）的构造。假设在参数值为 `lambda` 的情况下，有 `K_lambda` 个连接块，我们用 `A_1,A_2,..,A_K_lambda` 表示。这样我们可以重写（2）为如下（3）的形式。

$$\frac{1}{2} \sum_{i=1}^{K_\lambda} \sum_{j \in A_i} (y_j - \beta_{A_i})^2 + \lambda \sum_{i=1}^{K_\lambda - 1} (\beta_{A_i} - \beta_{A_{i+1}})_+. \tag{3}$$

上面的公式，对 `beta` 求偏导，可以得到下面的次梯度公式。通过这个公式即可以求得 `beta` 。

$$-\sum_{j \in A_i} y_j + |A_i| \hat{\beta}_{\lambda,A_i} + \lambda(s_i - s_{i-1}) = 0 \quad \text{for } i = 1, \ldots K_\lambda. \tag{4}$$

为了符合方便，令 `s_0 = s_K_lambda = 0` 。并且，

$$s_i = 1(\hat{\beta}_{\lambda,A_i} - \hat{\beta}_{\lambda,A_{i+1}} > 0)$$

现在假设，当 `lambda` 在一个区间内增长时，组 `A_1,A_2,...,A_K_lambda` 不会改变。我们可以通过相应的 `lambda` 区分（4）。

$$\frac{d\hat{\beta}_{\lambda,A_i}}{d\lambda} = \frac{s_{i-1} - s_i}{|A_i|}, \tag{5}$$

这个公式的值本身是一个常量，它意味着上式的 `beta` 是 `lambda` 的线性函数。

随着 `lambda` 的增长，方程（5）将连续的给出解决方案的斜率直到组 `A_1,A_2,...,A_K_lambda` 改变。更加引理1，只有两个组合并时，这才会发生。 `m_i` 表示斜率，那么对于每一个 `i=1,...,K_lambda - 1` ， `A_i` 和 `A_i+1` 合并之后得到的公式如下

$$t_{i,i+1} = \frac{\hat{\beta}_{\lambda,A_{i+1}} - \hat{\beta}_{\lambda,A_i}}{m_i - m_{i+1}} + \lambda, \tag{6}$$

因此我们可以一直移动，直到 `lambda` “下一个”值的到来

$$\lambda^* = \min_{i:\, t_{i,i+1} > \lambda} t_{i,i+1}, \tag{7}$$

并且合并 `A_i^star` 和 `A_i^star+1` ,其中

$$i^* = \operatorname*{argmin}_{i:\, t_{i,i+1} > \lambda} t_{i,i+1}. \tag{8}$$

注意，可能有超过一对组别到达了这个最小值，在这种情况下，会组合所有满足条件的组别。公式（7）和（8）成立的条件是 `t_i,i+1` 大于 `lambda` ，如果没有 `t_i,i+1` 大于 `lambda` ，说明没有组别可以合并，算法将会终止。

算法的流程如下：

- 初始时， `lambda=0` ， `K_lambda=n` , `A_i={i},i=1,2,...,n` 。对于每个i，解是 `beta_lambda,i = y_i`

- 重复下面过程

  1、通过公式（5）计算每个组的斜率 `m_i`

  2、通过公式（6）计算没对相邻组的碰撞次数 `t_i,i+1`

  3、如果 `t_i,i+1 < lambda` ，终止

  4、计算公式（7）中的临界点 `lambda^star` ,并根据斜率更新解

$$\hat{\beta}_{\lambda^*, A_i} = \hat{\beta}_{\lambda, A_i} + m_i \cdot (\lambda^* - \lambda)$$

对于每个 `i` ，根据公式（8）合并合适的组别（所以 `K_lambda^star = K_lambda - 1` ），并设置 `lambda = lambda^star` 。

4 源码分析

在 `1.6.x` 版本中，并没有实现近似保序回归，后续会实现。现在我们只介绍一般的保序回归算法实现。

4.1 实例

```
import org.apache.spark.mllib.regression.{IsotonicRegression, IsotonicRegressionModel}
val data = sc.textFile("data/mllib/sample_isotonic_regression_data.txt")
// 创建（label, feature, weight） tuples ，权重默认设置为1.0
val parsedData = data.map { line =>
  val parts = line.split(',').map(_.toDouble)
  (parts(0), parts(1), 1.0)
}
// Split data into training (60%) and test (40%) sets.
val splits = parsedData.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0)
val test = splits(1)
// Create isotonic regression model from training data.
// Isotonic parameter defaults to true so it is only shown for demonstration
val model = new IsotonicRegression().setIsotonic(true).run(training)
// Create tuples of predicted and real labels.
val predictionAndLabel = test.map { point =>
  val predictedLabel = model.predict(point._2)
  (predictedLabel, point._1)
}
// Calculate mean squared error between predicted and real labels.
val meanSquaredError = predictionAndLabel.map { case (p, l) => math.pow((p - l), 2) }.mean()
println("Mean Squared Error = " + meanSquaredError)
```

4.2 训练过程分析

`parallelPoolAdjacentViolators` 方法用于实现保序回归的训练。`parallelPoolAdjacentViolators` 方法的代码如下：

```
private def parallelPoolAdjacentViolators(
      input: RDD[(Double, Double, Double)]): Array[(Double, Double, Double)] = {
    val parallelStepResult = input
      //以（feature，label）为key进行排序
      .sortBy(x => (x._2, x._1))
      .glom()//合并不同分区的数据为一个数组
      .flatMap(poolAdjacentViolators)
      .collect()
      .sortBy(x => (x._2, x._1)) // Sort again because collect() doesn't promise ordering.
    poolAdjacentViolators(parallelStepResult)
  }
```

`parallelPoolAdjacentViolators` 方法的主要实现是 `poolAdjacentViolators` 方法，该方法主要的实现过程如下：

```
var i = 0
val len = input.length
while (i < len) {
     var j = i
     //找到破坏单调性的元祖的index
     while (j < len - 1 && input(j)._1 > input(j + 1)._1) {
       j = j + 1
     }
     // 如果没有找到违规点，移动到下一个数据点
     if (i == j) {
       i = i + 1
     } else {
       // 否则用pool方法处理违规的节点
       // 并且检查pool之后，之前处理过的节点是否违反了单调性约束
       while (i >= 0 && input(i)._1 > input(i + 1)._1) {
          pool(input, i, j)
          i = i - 1
       }
       i = j
     }
}
```

`pool` 方法的实现如下所示。

```
def pool(input: Array[(Double, Double, Double)], start: Int, end
: Int): Unit = {
      //取得i到j之间的元组组成的子序列
      val poolSubArray = input.slice(start, end + 1)
      //求子序列sum（label * w）之和
      val weightedSum = poolSubArray.map(lp => lp._1 * lp._3).su
m
      //求权重之和
      val weight = poolSubArray.map(_._3).sum
      var i = start
      //子区间的所有元组标签相同，即拥有相同的预测
      while (i <= end) {
        //修改标签值为两者之商
        input(i) = (weightedSum / weight, input(i)._2, input(i).
_3)
        i = i + 1
      }
}
```

经过上文的处理之后，`input` 根据中的 `label` 和 `feature` 均是按升序排列。对于拥有相同预测的点，我们只保留两个特征边界点。

```
val compressed = ArrayBuffer.empty[(Double, Double, Double)]
var (curLabel, curFeature, curWeight) = input.head
var rightBound = curFeature
def merge(): Unit = {
    compressed += ((curLabel, curFeature, curWeight))
    if (rightBound > curFeature) {
        compressed += ((curLabel, rightBound, 0.0))
    }
}
i = 1
while (i < input.length) {
    val (label, feature, weight) = input(i)
    if (label == curLabel) {
       //权重叠加
       curWeight += weight
       rightBound = feature
    } else {//如果标签不同，合并
       merge()
       curLabel = label
       curFeature = feature
       curWeight = weight
       rightBound = curFeature
    }
    i += 1
}
merge()
```

最后将训练的结果保存为模型。

```
//标签集
val predictions = if (isotonic) pooled.map(_._1) else pooled.map
(-_._1)
//特征集
val boundaries = pooled.map(_._2)
new IsotonicRegressionModel(boundaries, predictions, isotonic)
```

4.3 预测过程分析

```
def predict(testData: Double): Double = {
    def linearInterpolation(x1: Double, y1: Double, x2: Double,
y2: Double, x: Double): Double = {
      y1 + (y2 - y1) * (x - x1) / (x2 - x1)
    }
    //二分查找index
    val foundIndex = binarySearch(boundaries, testData)
    val insertIndex = -foundIndex - 1
    // Find if the index was lower than all values,
    // higher than all values, in between two values or exact ma
tch.
    if (insertIndex == 0) {
      predictions.head
    } else if (insertIndex == boundaries.length){
      predictions.last
    } else if (foundIndex < 0) {
      linearInterpolation(
        boundaries(insertIndex - 1),
        predictions(insertIndex - 1),
        boundaries(insertIndex),
        predictions(insertIndex),
        testData)
    } else {
      predictions(foundIndex)
    }
  }
```

当测试数据精确匹配一个边界，那么返回相应的特征。如果测试数据比所有边界都大或者小，那么分别返回第一个和最后一个特征。当测试数据位于两个边界之间，使用 `linearInterpolation` 方法计算特征。 这个方法是线性内插法。

聚类

聚类是一种无监督学习问题，它的目标就是基于相似度将相似的子集聚合在一起。聚类经常用于探索性研究或者作为分层有监督流程的一部分。
`spark.mllib` 包中支持下面的模型。

- k-means算法
- GMM（高斯混合模型）
- PIC（快速迭代聚类）
- LDA（隐式狄利克雷分布)
- 二分k-means算法
- 流式k-means算法

`k-means` 、 `k-means++` 以及 `k-means||` 算法分析

本文会介绍一般的 `k-means` 算法、 `k-means++` 算法以及基于 `k-means++` 算法的 `k-means||` 算法。在 `spark ml` ，已经实现了 `k-means` 算法以及 `k-means||` 算法。 本文首先会介绍这三个算法的原理，然后在了解原理的基础上分析 `spark` 中的实现代码。

1 `k-means` 算法原理分析

`k-means` 算法是聚类分析中使用最广泛的算法之一。它把 `n` 个对象根据它们的属性分为 `k` 个聚类以便使得所获得的聚类满足：同一聚类中的对象相似度较高；而不同聚类中的对象相似度较小。

`k-means` 算法的基本过程如下所示：

- （1）任意选择 `k` 个初始中心$c{1},c{2},...,c_{k}$ 。
- （2）计算 `X` 中的每个对象与这些中心对象的距离；并根据最小距离重新对相应对象进行划分；
- （3）重新计算每个中心对象C_{i}的值

$$C_i\colon\ c_i = \frac{1}{|C_i|}\sum\nolimits_{x \in C_i} x$$

- （4）计算标准测度函数，当满足一定条件，如函数收敛时，则算法终止；如果条件不满足则重复步骤（2），（3）。

1.1 `k-means` 算法的缺点

`k-means` 算法虽然简单快速，但是存在下面的缺点：

- 聚类中心的个数 `K` 需要事先给定，但在实际中 `K` 值的选定是非常困难的，很多时候我们并不知道给定的数据集应该分成多少个类别才最合适。

- `k-means` 算法需要随机地确定初始聚类中心，不同的初始聚类中心可能导致完全不同的聚类结果。

第一个缺陷我们很难在 `k-means` 算法以及其改进算法中解决，但是我们可以通过 `k-means++` 算法来解决第二个缺陷。

2 `k-means++` 算法原理分析

`k-means++` 算法选择初始聚类中心的基本原则是：初始的聚类中心之间的相互距离要尽可能的远。它选择初始聚类中心的步骤是：

- （1）从输入的数据点集合中随机选择一个点作为第一个聚类中心c_{1} ；
- （2）对于数据集中的每一个点 `x` ，计算它与最近聚类中心(指已选择的聚类中心)的距离 `D(x)` ，并根据概率选择新的聚类中心c_{i} 。
- （3）重复过程（2）直到找到k个聚类中心。

第(2)步中，依次计算每个数据点与最近的种子点（聚类中心）的距离，依次得到 `D(1)、D(2)、...、D(n)` 构成的集合 `D` ，其中 `n` 表示数据集的大小。
在 `D` 中，为了避免噪声，不能直接选取值最大的元素，应该选择值较大的元素，然后将其对应的数据点作为种子点。 如何选择值较大的元素呢，下面是 `spark` 中实现的思路。

- 求所有的距离和 `Sum(D(x))`
- 取一个随机值，用权重的方式来取计算下一个“种子点”。这个算法的实现是，先用 `Sum(D(x))` 乘以随机值 `Random` 得到值 `r` ，然后用 `currSum += D(x)` ，直到其 `currSum > r` ，此时的点就是下一个“种子点”。

为什么用这样的方式呢？我们换一种比较好理解的方式来说明。把集合 `D` 中的每个元素 `D(x)` 想象为一根线 `L(x)` ，线的长度就是元素的值。将这些线依次按照 `L(1)、L(2)、...、L(n)` 的顺序连接起来，组成长线 `L` 。 `L(1)、L(2)、…、L(n)` 称为 `L` 的子线。 根据概率的相关知识，如果我们在 `L` 上随机选择一个点，那么这个点所在的子线很有可能是比较长的子线，而这个子线对应的数据点就可以作为种子点。

2.1 `k-means++` 算法的缺点

虽然 `k-means++` 算法可以确定地初始化聚类中心，但是从可扩展性来看，它存在一个缺点，那就是它内在的有序性特性：下一个中心点的选择依赖于已经选择的中心点。 针对这种缺陷， `k-means||` 算法提供了解决方法。

3 `k-means||` 算法原理分析

`k-means||` 算法是在 `k-means++` 算法的基础上做的改进，和 `k-means++` 算法不同的是，它采用了一个采样因子 `l` ，并且 `l=A(k)` ，在 `spark` 的实现中 `l=2k` ，。这个算法首先如 `k-means++` 算法一样，随机选择一个初始中心， 然后计算选定初始中心确定之后的初始花费ψ(指与最近中心点的距离)。之后处理$log(\psi)$次迭代，在每次迭代中，给定当前中心集，通过概率$ld^{2}(x,C)/\phi{X}(C)$来 抽样 `x` ，将选定的 `x` 添加到初始化中心集中，并且更新$\phi{X}(C)$。该算法的步骤如下图所示：

1: $\mathcal{C} \leftarrow$ sample a point uniformly at random from X
2: $\psi \leftarrow \phi_X(\mathcal{C})$
3: **for** $O(\log \psi)$ times **do**
4: $\quad \mathcal{C}' \leftarrow$ sample each point $x \in X$ independently with probability $p_x = \frac{\ell \cdot d^2(x,\mathcal{C})}{\phi_X(\mathcal{C})}$
5: $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
6: **end for**
7: For $x \in \mathcal{C}$, set w_x to be the number of points in X closer to x than any other point in $\mathcal{C}$
8: Recluster the weighted points in $\mathcal{C}$ into k clusters

第1步随机初始化一个中心点，第2-6步计算出满足概率条件的多个候选中心点 `C` ，候选中心点的个数可能大于 `k` 个，所以通过第7-8步来处理。第7步给 `C` 中所有点赋予一个权重值w_{x} ，这个权重值表示距离 `x` 点最近的点的个数。 第8步使用本地 `k-means++` 算法聚类出这些候选点的 `k` 个聚类中心。在 `spark` 的源码中，迭代次数是人为设定的，默认是5。

该算法与 `k-means++` 算法不同的地方是它每次迭代都会抽样出多个中心点而不是一个中心点，且每次迭代不互相依赖，这样我们可以并行的处理这个迭代过程。由于该过程产生出来的中心点的数量远远小于输入数据点的数量， 所以第8步可以通过本地 `k-means++` 算法很快的找出 `k` 个初始化中心点。何为本地 `k-means++` 算法？就是运行在单个机器节点上的 `k-means++` 。

下面我们详细分析上述三个算法的代码实现。

4 源代码分析

在 `spark` 中， `org.apache.spark.mllib.clustering.KMeans` 文件实现了 `k-means` 算法以及 `k-means||` 算法， `org.apache.spark.mllib.clustering.LocalKMeans` 文件实现了 `k-means++` 算法。 在分步骤分析 `spark` 中的源码之前我们先来了解 `KMeans` 类中参数的含义。

```
class KMeans private (
    private var k: Int,//聚类个数
    private var maxIterations: Int,//迭代次数
    private var runs: Int,//运行kmeans算法的次数
    private var initializationMode: String,//初始化模式
    private var initializationSteps: Int,//初始化步数
    private var epsilon: Double,//判断kmeans算法是否收敛的阈值
    private var seed: Long)
```

在上面的定义中， `k` 表示聚类的个数， `maxIterations` 表示最大的迭代次数， `runs` 表示运行 `KMeans` 算法的次数，在 `spark 2.0。0` 开始，该参数已经不起作用了。为了更清楚的理解算法我们可以认为它为1。 `initializationMode` 表示初始化模式，有两种选择：随机初始化和通过 `k-means||` 初始化，默认是通过 `k-means||` 初始化。 `initializationSteps` 表示通过 `k-means||` 初始化时的迭代步骤，默认是5，这是 `spark` 实现与第三章的算法步骤不一样的地方，这里迭代次数人为指定， 而第三章的算法是根据距离得到的迭代次数，为 `log(phi)` 。 `epsilon` 是判断算法是否已经收敛的阈值。

下面将分步骤分析 `k-means` 算法、 `k-means||` 算法的实现过程。

4.1 处理数据，转换为 `VectorWithNorm` 集。

```
//求向量的二范式，返回double值
val norms = data.map(Vectors.norm(_, 2.0))
norms.persist()
val zippedData = data.zip(norms).map { case (v, norm) =>
   new VectorWithNorm(v, norm)
}
```

4.2 初始化中心点。

初始化中心点根据 `initializationMode` 的值来判断，如果 `initializationMode` 等于 `KMeans.RANDOM` ，那么随机初始化 `k` 个中心点，否则使用 `k-means||` 初始化 `k` 个中心点。

```
val centers = initialModel match {
      case Some(kMeansCenters) => {
        Array(kMeansCenters.clusterCenters.map(s => new VectorWi
thNorm(s)))
      }
      case None => {
        if (initializationMode == KMeans.RANDOM) {
          initRandom(data)
        } else {
          initKMeansParallel(data)
        }
      }
    }
```

- （1）随机初始化中心点。

随机初始化 `k` 个中心点很简单，具体代码如下：

```
private def initRandom(data: RDD[VectorWithNorm])
  : Array[Array[VectorWithNorm]] = {
    //采样固定大小为k的子集
    //这里run表示我们运行的KMeans算法的次数，默认为1，以后将停止提供该参数
    val sample = data.takeSample(true, runs * k, new XORShiftRan
dom(this.seed).nextInt()).toSeq
    //选取k个初始化中心点
    Array.tabulate(runs)(r => sample.slice(r * k, (r + 1) * k).m
ap { v =>
      new VectorWithNorm(Vectors.dense(v.vector.toArray), v.norm
)
    }.toArray)
  }
```

- （2）通过 `k-means||` 初始化中心点。

相比于随机初始化中心点，通过 `k-means||` 初始化 `k` 个中心点会麻烦很多，它需要依赖第三章的原理来实现。它的实现方法是 `initKMeansParallel` 。下面按照第三章的实现步骤来分析。

- 第一步，我们要随机初始化第一个中心点。

```
//初始化第一个中心点
val seed = new XORShiftRandom(this.seed).nextInt()
val sample = data.takeSample(true, runs, seed).toSeq
val newCenters = Array.tabulate(runs)(r => ArrayBuffer(sample(r)
.toDense))
```

- 第二步，通过已知的中心点，循环迭代求得其它的中心点。

```
var step = 0
while (step < initializationSteps) {
    val bcNewCenters = data.context.broadcast(newCenters)
    val preCosts = costs
    //每个点距离最近中心的代价
    costs = data.zip(preCosts).map { case (point, cost) =>
          Array.tabulate(runs) { r =>
            //pointCost获得与最近中心点的距离
            //并与前一次迭代的距离对比取更小的那个
            math.min(KMeans.pointCost(bcNewCenters.value(r), poi
nt), cost(r))
          }
        }.persist(StorageLevel.MEMORY_AND_DISK)
    //所有点的代价和
    val sumCosts = costs.aggregate(new Array[Double](runs))(
          //分区内迭代
          seqOp = (s, v) => {
            // s += v
            var r = 0
            while (r < runs) {
              s(r) += v(r)
              r += 1
            }
```

```
            s
          },
          //分区间合并
          combOp = (s0, s1) => {
            // s0 += s1
            var r = 0
            while (r < runs) {
              s0(r) += s1(r)
              r += 1
            }
            s0
          }
        )
    //选择满足概率条件的点
    val chosen = data.zip(costs).mapPartitionsWithIndex { (index
, pointsWithCosts) =>
        val rand = new XORShiftRandom(seed ^ (step << 16) ^ inde
x)
        pointsWithCosts.flatMap { case (p, c) =>
          val rs = (0 until runs).filter { r =>
            //此处设置l=2k
            rand.nextDouble() < 2.0 * c(r) * k / sumCosts(r)
          }
          if (rs.length > 0) Some(p, rs) else None
        }
      }.collect()
      mergeNewCenters()
      chosen.foreach { case (p, rs) =>
        rs.foreach(newCenters(_) += p.toDense)
      }
      step += 1
}
```

在这段代码中，我们并没有选择使用 `log(pha)` 的大小作为迭代的次数，而是直接使用了人为确定的 `initializationSteps` ，这是与论文中不一致的地方。在迭代内部我们使用概率公式

$$p_x = \frac{\ell \cdot d^2(x, C)}{\phi_X(C)}$$

来计算满足要求的点，其中， `l=2k` 。公式的实现如代码 `rand.nextDouble() < 2.0 * c(r) * k / sumCosts(r)` 。 `sumCosts` 表示所有点距离它所属类别的中心点的欧式距离之和。 上述代码通过 `aggregate` 方法并行计算获得该值。

- 第三步，求最终的k个点。

通过以上步骤求得的候选中心点的个数可能会多于 `k` 个，这样怎么办呢？我们给每个中心点赋一个权重，权重值是数据集中属于该中心点所在类别的数据点的个数。 然后我们使用本地 `k-means++` 来得到这 `k` 个初始化点。具体的实现代码如下：

```
val bcCenters = data.context.broadcast(centers)
    //计算权重值，即各中心点所在类别的个数
    val weightMap = data.flatMap { p =>
      Iterator.tabulate(runs) { r =>
        ((r, KMeans.findClosest(bcCenters.value(r), p)._1), 1.0)
      }
    }.reduceByKey(_ + _).collectAsMap()
    //最终的初始化中心
    val finalCenters = (0 until runs).par.map { r =>
      val myCenters = centers(r).toArray
      val myWeights = (0 until myCenters.length).map(i => weight
Map.getOrElse((r, i), 0.0)).toArray
      LocalKMeans.kMeansPlusPlus(r, myCenters, myWeights, k, 30)
    }
```

上述代码的关键点时通过本地 `k-means++` 算法求最终的初始化点。它是通过 `LocalKMeans.kMeansPlusPlus` 来实现的。它使用 `k-means++` 来处理。

```
// 初始化一个中心点
centers(0) = pickWeighted(rand, points, weights).toDense
//
for (i <- 1 until k) {
      // 根据概率比例选择下一个中心点
      val curCenters = centers.view.take(i)
      //每个点的权重与距离的乘积和
      val sum = points.view.zip(weights).map { case (p, w) =>
        w * KMeans.pointCost(curCenters, p)
      }.sum
      //取随机值
      val r = rand.nextDouble() * sum
      var cumulativeScore = 0.0
      var j = 0
      //寻找概率最大的点
      while (j < points.length && cumulativeScore < r) {
        cumulativeScore += weights(j) * KMeans.pointCost(curCent
ers, points(j))
        j += 1
      }
      if (j == 0) {
        centers(i) = points(0).toDense
      } else {
        centers(i) = points(j - 1).toDense
      }
}
```

上述代码中， `points` 指的是候选的中心点， `weights` 指这些点相应地权重。寻找概率最大的点的方式就是第二章提到的方式。初始化 `k` 个中心点后， 就可以通过一般的 `k-means` 流程来求最终的 `k` 个中心点了。具体的过程4.3会讲到。

4.3 确定数据点所属类别

找到中心点后，我们就需要根据距离确定数据点的聚类，即数据点和哪个中心点最近。具体代码如下：

```
    // 找到每个聚类中包含的点距离中心点的距离和以及这些点的个数
    val totalContribs = data.mapPartitions { points =>
      val thisActiveCenters = bcActiveCenters.value
      val runs = thisActiveCenters.length
      val k = thisActiveCenters(0).length
      val dims = thisActiveCenters(0)(0).vector.size

      val sums = Array.fill(runs, k)(Vectors.zeros(dims))
      val counts = Array.fill(runs, k)(0L)

      points.foreach { point =>
        (0 until runs).foreach { i =>
          //找到离给定点最近的中心以及相应的欧几里得距离
          val (bestCenter, cost) = KMeans.findClosest(thisActiveCenters(i), point)
          costAccums(i) += cost
          //距离和
          val sum = sums(i)(bestCenter)
          //y += a * x
          axpy(1.0, point.vector, sum)
          //点数量
          counts(i)(bestCenter) += 1
        }
      }

      val contribs = for (i <- 0 until runs; j <- 0 until k) yield {
        ((i, j), (sums(i)(j), counts(i)(j)))
      }
      contribs.iterator
    }.reduceByKey(mergeContribs).collectAsMap()
```

4.4 重新确定中心点

找到类别中包含的数据点以及它们距离中心点的距离，我们可以重新计算中心点。代码如下：

```
//更新中心点
for ((run, i) <- activeRuns.zipWithIndex) {
    var changed = false
    var j = 0
    while (j < k) {
        val (sum, count) = totalContribs((i, j))
        if (count != 0) {

            //x = a * x，求平均距离即sum/count
            scal(1.0 / count, sum)

            val newCenter = new VectorWithNorm(sum)
            //如果新旧两个中心点的欧式距离大于阈值
            if (KMeans.fastSquaredDistance(newCenter, centers(ru
n)(j)) > epsilon * epsilon) {
              changed = true
            }
            centers(run)(j) = newCenter
        }
        j += 1
    }
    if (!changed) {
        active(run) = false
        logInfo("Run " + run + " finished in " + (iteration + 1)
 + " iterations")
    }
    costs(run) = costAccums(i).value
}
```

5 参考文献

- 【1】Bahman Bahmani,Benjamin Moseley,Andrea Vattani.Scalable K-Means++
- 【2】David Arthur and Sergei Vassilvitskii.k-means++: The Advantages of Careful Seeding

高斯混合模型

> 现有的高斯模型有单高斯模型（ `SGM` ）和高斯混合模型（ `GMM` ）两种。从几何上讲，单高斯分布模型在二维空间上近似于椭圆，在三维空间上近似于椭球。 在很多情况下，属于同一类别的样本点并不满足“椭圆”分布的特性，所以我们需要引入混合高斯模型来解决这种情况。

1 单高斯模型

多维变量 `X` 服从高斯分布时，它的概率密度函数 `PDF` 定义如下：

$$N(x;\mu,\Sigma)=\frac{1}{(2\pi)^{D/2}}\frac{1}{(|\Sigma|)^{1/2}}\exp[-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)]$$

在上述定义中, `x` 是维数为 `D` 的样本向量， `mu` 是模型期望， `sigma` 是模型协方差。对于单高斯模型，可以明确训练样本是否属于该高斯模型，所以我们经常将 `mu` 用训练样本的均值代替，将 `sigma` 用训练样本的协方差代替。 假设训练样本属于类别 `C` ，那么上面的定义可以修改为下面的形式：

$$N(x;C)=\frac{1}{(2\pi)^{D/2}}\frac{1}{(|\Sigma|)^{1/2}}\exp[-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)]$$

这个公式表示样本属于类别 `C` 的概率。我们可以根据定义的概率阈值来判断样本是否属于某个类别。

2 高斯混合模型

高斯混合模型，顾名思义，就是数据可以看作是从多个高斯分布中生成出来的。从中心极限定理可以看出，高斯分布这个假设其实是比较合理的。 为什么我们要假设数据是由若干个高斯分布组合而成的，而不假设是其他分布呢？实际上不管是什么分布，只 `K` 取得足够大，这个 `XX Mixture Model` 就会变得足够复杂，就可以用来逼近任意连续的概率密度分布。只是因为高斯函数具有良好的计算性能，所 `GMM` 被广泛地应用。

每个 `GMM` 由 `K` 个高斯分布组成，每个高斯分布称为一个组件（ `Component` ），这些组件线性加成在一起就组成了 `GMM` 的概率密度函数（1）：

$$p(x) = \sum_{k=1}^{K} p(k)p(x|k) = \sum_{k=1}^{K} \pi_k N(x|\mu_k, \Sigma_k)$$

根据上面的式子，如果我们要从 `GMM` 分布中随机地取一个点，需要两步：

- 随机地在这 `K` 个组件之中选一个，每个组件被选中的概率实际上就是它的系数 `pi_k` ；
- 选中了组件之后，再单独地考虑从这个组件的分布中选取一个点。

怎样用 `GMM` 来做聚类呢？其实很简单，现在我们有了数据，假定它们是由 `GMM` 生成出来的，那么我们只要根据数据推出 `GMM` 的概率分布来就可以了，然后 `GMM` 的 `K` 个组件实际上就对应了 `K` 个聚类了。 在已知概率密度函数的情况下，要估计其中的参数的过程被称作“参数估计”。

我们可以利用最大似然估计来确定这些参数， `GMM` 的似然函数（2）如下：

$$\sum_{i=1}^{N} log\left\{\sum_{k=1}^{K} \pi_k N(x|\mu_k, \Sigma_k)\right\}$$

可以用 `EM` 算法来求解这些参数。 `EM` 算法求解的过程如下：

- 1 **E-**步。求数据点由各个组件生成的概率（并不是每个组件被选中的概率）。对于每个数据x_{i}来说，它由第 `k` 个组件生成的概率为公式（3）：

$$\gamma(i, k) = \frac{\pi_k N(x_i|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j N(x_i|\mu_j, \Sigma_{kj})}$$

在上面的概率公式中，我们假定 `mu` 和 `sigma` 均是已知的，它们的值来自于初始化值或者上一次迭代。

- 2 **M-**步。估计每个组件的参数。由于每个组件都是一个标准的高斯分布，可以很容易分布求出最大似然所对应的参数值，分别如下公式（4），（5），（6），

（7）：

$$N_k = \sum_{i=1}^{N} \gamma(\mathrm{i},\mathrm{k})$$

$$\pi_k = \frac{N_k}{N}$$

$$\mu_k = \frac{1}{N_k}\sum_{i=1}^{N} \gamma(\mathrm{i},\mathrm{k})x_i$$

$$\Sigma_k = \frac{1}{N_k}\sum_{i=1}^{N} \gamma(\mathrm{i},\mathrm{k})(x_i-\mu_k)(x_i-\mu_k)^T$$

3 源码分析

3.1 实例

在分析源码前，我们还是先看看高斯混合模型如何使用。

```
import org.apache.spark.mllib.clustering.GaussianMixture
import org.apache.spark.mllib.clustering.GaussianMixtureModel
import org.apache.spark.mllib.linalg.Vectors
// 加载数据
val data = sc.textFile("data/mllib/gmm_data.txt")
val parsedData = data.map(s => Vectors.dense(s.trim.split(' ').m
ap(_.toDouble))).cache()
// 使用高斯混合模型聚类
val gmm = new GaussianMixture().setK(2).run(parsedData)
// 保存和加载模型
gmm.save(sc, "myGMMModel")
val sameModel = GaussianMixtureModel.load(sc, "myGMMModel")
// 打印参数
for (i <- 0 until gmm.k) {
  println("weight=%f\nmu=%s\nsigma=\n%s\n" format
    (gmm.weights(i), gmm.gaussians(i).mu, gmm.gaussians(i).sigma
))
}
```

由上面的代码我们可以知道，使用高斯混合模型聚类使用到了 `GaussianMixture` 类中的 `run` 方法。下面我们直接进入 `run` 方法，分析它的实现。

3.2 高斯混合模型的实现

3.2.1 初始化

在 `run` 方法中，程序所做的第一步就是初始化权重（上文中介绍的 `pi` ）及其相对应的高斯分布。

```
val (weights, gaussians) = initialModel match {
      case Some(gmm) => (gmm.weights, gmm.gaussians)
      case None => {
        val samples = breezeData.takeSample(withReplacement = true, k * nSamples, seed)
        (Array.fill(k)(1.0 / k), Array.tabulate(k) { i =>
          val slice = samples.view(i * nSamples, (i + 1) * nSamples)
          new MultivariateGaussian(vectorMean(slice), initCovariance(slice))
        })
      }
    }
```

在上面的代码中，当 `initialModel` 为空时，用所有值均为 `1.0/k` 的数组初始化权重，用值为 `MultivariateGaussian` 对象的数组初始化所有的高斯分布（即上文中提到的组件）。每一个 `MultivariateGaussian` 对象都由从数据集中抽样的子集计算而来。这里用样本数据的均值和方差初始化 `MultivariateGaussian` 的 `mu` 和 `sigma`。

```
//计算均值
private def vectorMean(x: IndexedSeq[BV[Double]]): BDV[Double] =
 {
    val v = BDV.zeros[Double](x(0).length)
    x.foreach(xi => v += xi)
    v / x.length.toDouble
  }
//计算方差
private def initCovariance(x: IndexedSeq[BV[Double]]): BreezeMatrix[Double] = {
    val mu = vectorMean(x)
    val ss = BDV.zeros[Double](x(0).length)
    x.foreach(xi => ss += (xi - mu) :^ 2.0)
    diag(ss / x.length.toDouble)
  }
```

3.2.2 EM算法求参数

初始化后，就可以使用 `EM` 算法迭代求似然函数中的参数。迭代结束的条件是迭代次数达到了我们设置的次数或者两次迭代计算的对数似然值之差小于阈值。

```
while (iter < maxIterations && math.abs(llh-llhp) > convergenceTol)
```

在迭代内部，就可以按照 `E-步` 和 `M-步` 来更新参数了。

- **E-**步：更新参数 `gamma`

```
val compute = sc.broadcast(ExpectationSum.add(weights, gaussians)_)
val sums = breezeData.aggregate(ExpectationSum.zero(k, d))(compute.value, _ += _)
```

我们先要了解 `ExpectationSum` 以及 `add` 方法的实现。

```
private class ExpectationSum(
    var logLikelihood: Double,
    val weights: Array[Double],
    val means: Array[BDV[Double]],
    val sigmas: Array[BreezeMatrix[Double]]) extends Serializable
```

`ExpectationSum` 是一个聚合类，它表示部分期望结果：主要包含对数似然值，权重值（第二章中介绍的 `pi` ），均值，方差。 `add` 方法的实现如下：

```
def add( weights: Array[Double],dists: Array[MultivariateGaussian
])
      (sums: ExpectationSum, x: BV[Double]): ExpectationSum = {
    val p = weights.zip(dists).map {
      //计算pi_i * N(x)
      case (weight, dist) => MLUtils.EPSILON + weight * dist.pdf
(x)
    }
    val pSum = p.sum
    sums.logLikelihood += math.log(pSum)
    var i = 0
    while (i < sums.k) {
      p(i) /= pSum
      sums.weights(i) += p(i)
      sums.means(i) += x * p(i)
      //A := alpha * x * x^T^ + A
      BLAS.syr(p(i), Vectors.fromBreeze(x),
        Matrices.fromBreeze(sums.sigmas(i)).asInstanceOf[DenseMa
trix])
      i = i + 1
    }
    sums
  }
```

从上面的实现我们可以看出，最终， `logLikelihood` 表示公式**(2)**中的对数似然。 `p` 和 `weights` 分别表示公式**(3)**中的 `gamma` 和 `pi` ， `means` 表示公式**(6)**中的求和部分， `sigmas` 表示公式**(7)**中的求和部分。

调用 `RDD` 的 `aggregate` 方法，我们可以基于所有给定数据计算上面的值。利用计算的这些新值，我们可以在 `M-步` 中更新 `mu` 和 `sigma` 。

- **M-**步：更新参数 `mu` 和 `sigma`

```
var i = 0
while (i < k) {
   val (weight, gaussian) =
      updateWeightsAndGaussians(sums.means(i), sums.sigmas(i),
sums.weights(i), sumWeights)
   weights(i) = weight
   gaussians(i) = gaussian
   i = i + 1
}
private def updateWeightsAndGaussians(
     mean: BDV[Double],
     sigma: BreezeMatrix[Double],
     weight: Double,
     sumWeights: Double): (Double, MultivariateGaussian) = {
   //  mean/weight
   val mu = (mean /= weight)
   // -weight * mu * mut +sigma
   BLAS.syr(-weight, Vectors.fromBreeze(mu),
     Matrices.fromBreeze(sigma).asInstanceOf[DenseMatrix])
   val newWeight = weight / sumWeights
   val newGaussian = new MultivariateGaussian(mu, sigma / weigh
t)
   (newWeight, newGaussian)
 }
```

基于**E-**步计算出来的值，根据公式**(6)**，我们可以通过 `(mean /= weight)` 来更新 `mu` ；根据公式**(7)**，我们可以通过 `BLAS.syr()` 来更新 `sigma` ；同时，根据公式**(5)**， 我们可以通过 `weight / sumWeights` 来计算 `pi` 。

迭代执行以上的**E-**步和**M-**步，到达一定的迭代数或者对数似然值变化较小后，我们停止迭代。这时就可以获得聚类后的参数了。

3.3 多元高斯模型中相关方法介绍

在上面的求参代码中，我们用到了 `MultivariateGaussian` 以及 `MultivariateGaussian` 中的部分方法，如 `pdf` 。 `MultivariateGaussian` 定义如下：

```
class MultivariateGaussian @Since("1.3.0") (
    @Since("1.3.0") val mu: Vector,
    @Since("1.3.0") val sigma: Matrix) extends Serializable
```

`MultivariateGaussian` 包含一个向量 `mu` 和一个矩阵 `sigma` ，分别表示期望和协方差。 `MultivariateGaussian` 最重要的方法是 `pdf` ，顾名思义就是计算给定数据的概率密度函数。它的实现如下：

```
private[mllib] def pdf(x: BV[Double]): Double = {
    math.exp(logpdf(x))
}
private[mllib] def logpdf(x: BV[Double]): Double = {
    val delta = x - breezeMu
    val v = rootSigmaInv * delta
    u + v.t * v * -0.5
 }
```

上面的 `rootSigmaInv` 和 `u` 通过方法 `calculateCovarianceConstants` 计算。根据公式**(1)**，这个概率密度函数的计算需要计算 `sigma` 的行列式以及逆。

```
sigma = U * D * U.t
inv(Sigma) = U * inv(D) * U.t = (D^{-1/2}^ * U.t).t * (D^{-1/2}^
 * U.t)
-0.5 * (x-mu).t * inv(Sigma) * (x-mu) = -0.5 * norm(D^{-1/2}^ * U
.t  * (x-mu))^2^
```

这里， `U` 和 `D` 是奇异值分解得到的子矩阵。 `calculateCovarianceConstants` 具体的实现代码如下：

```
private def calculateCovarianceConstants: (DBM[Double], Double)
= {
    val eigSym.EigSym(d, u) = eigSym(sigma.toBreeze.toDenseMatri
x) // sigma = u * diag(d) * u.t
    val tol = MLUtils.EPSILON * max(d) * d.length
    try {
      //所有非0奇异值的对数和
      val logPseudoDetSigma = d.activeValuesIterator.filter(_ >
tol).map(math.log).sum
      //通过求非负值的倒数平方根，计算奇异值对角矩阵的根伪逆矩阵
      val pinvS = diag(new DBV(d.map(v => if (v > tol) math.sqrt(
1.0 / v) else 0.0).toArray))
      (pinvS * u.t, -0.5 * (mu.size * math.log(2.0 * math.Pi) +
logPseudoDetSigma))
    } catch {
      case uex: UnsupportedOperationException =>
        throw new IllegalArgumentException("Covariance matrix ha
s no non-zero singular values")
    }
  }
```

上面的代码中，`eigSym` 用于分解 `sigma` 矩阵。

4 参考文献

【1】漫谈 Clustering (3): Gaussian Mixture Model

快速迭代聚类

1 谱聚类算法的原理

在分析快速迭代聚类之前，我们先来了解一下谱聚类算法。谱聚类算法是建立在谱图理论的基础上的算法，与传统的聚类算法相比，它能在任意形状的样本空间上聚类且能够收敛到全局最优解。 谱聚类算法的主要思想是将聚类问题转换为无向图的划分问题。

- 首先，数据点被看做一个图的顶点 `v` ，两数据的相似度看做图的边，边的集合由$E=A_{ij}$表示，由此构造样本数据集的相似度矩阵 `A` ，并求出拉普拉斯矩阵 `L` 。
- 其次，根据划分准则使子图内部相似度尽量大，子图之间的相似度尽量小，计算出L的特征值和特征向量
- 最后，选择k个不同的特征向量对数据点聚类

那么如何求拉普拉斯矩阵呢？

将相似度矩阵 `A` 的每行元素相加就可以得到该顶点的度，我们定义以度为对角元素的对角矩阵称为度矩阵 `D` 。可以通过 `A` 和 `D` 来确定拉普拉斯矩阵。拉普拉斯矩阵分为规范和非规范两种，规范的拉普拉斯矩阵 表示为 `L=D-A` ，非规范的拉普拉斯矩阵表示为$L=I-D^{-1}A$ 。

谱聚类算法的一般过程如下：

- （1）输入待聚类的数据点集以及聚类数 `k` ；
- （2）根据相似性度量构造数据点集的拉普拉斯矩阵 `L` ；
- （3）选取L的前 `k` 个（默认从小到大,这里的 `k` 和聚类数可以不一样）特征值和特征向量，构造特征向量空间（这实际上是一个降维的过程）；
- （4）使用传统方法对特征向量聚类，并对应于原始数据的聚类。

谱聚类算法和传统的聚类方法（例如 `K-means` ）比起来有不少优点：

- 和 `K-medoids` 类似，谱聚类只需要数据之间的相似度矩阵就可以了，而不必像 `K-means` 那样要求数据必须是 `N` 维欧氏空间中的向量。

- 由于抓住了主要矛盾，忽略了次要的东西，因此比传统的聚类算法更加健壮一些，对于不规则的误差数据不是那么敏感，而且性能也要好一些。

- 计算复杂度比 `K-means` 要小，特别是在像文本数据或者平凡的图像数据这样维度非常高的数据上运行的时候。

快速迭代算法和谱聚类算法都是将数据点嵌入到由相似矩阵推导出来的低维子空间中，然后直接或者通过 `k-means` 算法产生聚类结果，但是快速迭代算法有不同的地方。下面重点了解快速迭代算法的原理。

2 快速迭代算法的原理

在快速迭代算法中，我们构造另外一个矩阵$W=D^{-1}A$,同第一章做比对，我们可以知道 `W` 的最大特征向量就是拉普拉斯矩阵 `L` 的最小特征向量。 我们知道拉普拉斯矩阵有一个特性：第二小特征向量（即第二小特征值对应的特征向量）定义了图最佳划分的一个解，它可以近似最大化划分准则。更一般的， `k` 个最小的特征向量所定义的子空间很适合去划分图。 因此拉普拉斯矩阵第二小、第三小直到第 `k` 小的特征向量可以很好的将图 `W` 划分为 `k` 个部分。

注意，矩阵 `L` 的 `k` 个最小特征向量也是矩阵 `W` 的 `k` 个最大特征向量。计算一个矩阵最大的特征向量可以通过一个简单的方法来求得，那就是快速迭代（即 `PI` ）。 `PI` 是一个迭代方法，它以任意的向量v^{0}作为起始，依照下面的公式循环进行更新。

$$v^{t+1} = cWv^{t}$$

在上面的公式中， `c` 是标准化常量，是为了避免v^{t}产生过大的值，这里$c=||Wv^{t}||_{1}$ 。在大多数情况下，我们只关心第 `k` （k不为1）大的特征向量，而不关注最大的特征向量。 这是因为最大的特征向量是一个常向量：因为 `W` 每一行的和都为1。

快速迭代的收敛性在文献【1】中有详细的证明，这里不再推导。

快速迭代算法的一般步骤如下：

Input: A row-normalized affinity matrix W and the number of clusters k
Pick an initial vector $\mathbf{v}^0$
repeat
  Set $\mathbf{v}^{t+1} \leftarrow \frac{W\mathbf{v}^t}{\|W\mathbf{v}^t\|_1}$ and $\boldsymbol{\delta}^{t+1} \leftarrow |\mathbf{v}^{t+1} - \mathbf{v}^t|$.
  Increment t
until $|\boldsymbol{\delta}^t - \boldsymbol{\delta}^{t-1}| \simeq \mathbf{0}$
Use k-means to cluster points on $\mathbf{v}^t$
Output: Clusters $C_1, C_2, ..., C_k$

在上面的公式中，输入矩阵 `W` 根据$W=D^{-1}A$来计算。

3 快速迭代算法的源码实现

在 `spark` 中，文件 `org.apache.spark.mllib.clustering.PowerIterationClustering` 实现了快速迭代算法。我们从官方给出的例子出发来分析快速迭代算法的实现。

```
import org.apache.spark.mllib.clustering.{PowerIterationClustering, PowerIterationClusteringModel}
import org.apache.spark.mllib.linalg.Vectors
// 加载和切分数据
val data = sc.textFile("data/mllib/pic_data.txt")
val similarities = data.map { line =>
  val parts = line.split(' ')
  (parts(0).toLong, parts(1).toLong, parts(2).toDouble)
}
// 使用快速迭代算法将数据分为两类
val pic = new PowerIterationClustering()
  .setK(2)
  .setMaxIterations(10)
val model = pic.run(similarities)
//打印出所有的簇
model.assignments.foreach { a =>
  println(s"${a.id} -> ${a.cluster}")
}
```

在上面的例子中，我们知道数据分为三列，分别是起始id，目标id，以及两者的相似度，这里的 `similarities` 代表前面章节提到的矩阵 `A` 。有了数据之后，我们通过 `PowerIterationClustering` 的 `run` 方法来训练模型。`PowerIterationClustering` 类有三个参数：

- `k` ：聚类数
- `maxIterations` ：最大迭代数
- `initMode` ：初始化模式。初始化模式分为 `Random` 和 `Degree` 两种，针对不同的模式对数据做不同的初始化操作

  下面分步骤介绍 `run` 方法的实现。

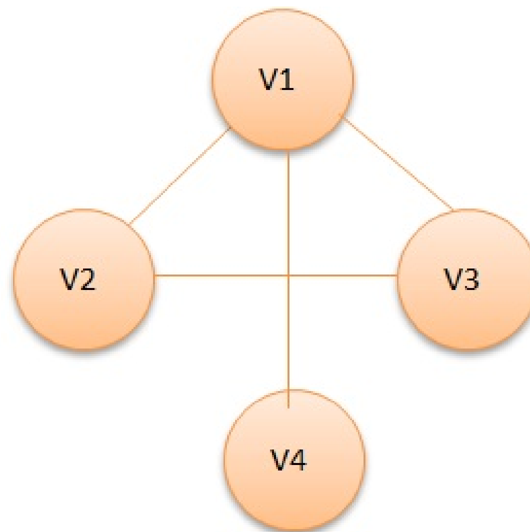
- （**1**）标准化相似度矩阵 **`A`** 到矩阵 **`W`**

```
def normalize(similarities: RDD[(Long, Long, Double)]): Graph[Do
uble, Double] = {
    //获得所有的边
    val edges = similarities.flatMap { case (i, j, s) =>
      //相似度值必须非负
      if (s < 0.0) {
        throw new SparkException("Similarity must be nonnegative
 but found s($i, $j) = $s.")
      }
      if (i != j) {
        Seq(Edge(i, j, s), Edge(j, i, s))
      } else {
        None
      }
    }
    //根据edges信息构造图，顶点的特征值默认为0
    val gA = Graph.fromEdges(edges, 0.0)
    //计算从顶点的出发的边的相似度之和，在这里称为度
    val vD = gA.aggregateMessages[Double](
      sendMsg = ctx => {
        ctx.sendToSrc(ctx.attr)
      },
      mergeMsg = _ + _,
      TripletFields.EdgeOnly)
    //计算得到W , W=A/D
    GraphImpl.fromExistingRDDs(vD, gA.edges)
      .mapTriplets(
        //gAi/vDi
        //使用边的权重除以起始点的度
        e => e.attr / math.max(e.srcAttr, MLUtils.EPSILON),
        TripletFields.Src)
  }
```

上面的代码首先通过边集合构造图 `gA` ,然后使用 `aggregateMessages` 计算每个顶点的度（即所有从该顶点出发的边的相似度之和），构造出 `VertexRDD` 。最后使用现有的 `VertexRDD` 和 `EdgeRDD` ，相继通过 `fromExistingRDDs` 和 `mapTriplets` 方法计算得到最终的图 `W` 。

在 `mapTriplets` 方法中，对每一个 `EdgeTriplet` ，使用相似度除以出发顶点的度（为什么相除？对角矩阵的逆矩阵是各元素取倒数，$W=D^{-1}A$就可以通过元素相除得到）。

下面举个例子来说明这个步骤。假设有 `v1,v2,v3,v4` 四个点，它们之间的关系如下图所示，并且假设点与点之间的相似度均设为1。



通过该图，我们可以得到相似度矩阵 `A` 和度矩阵 `D` ，他们分别如下所示。

$$A=\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&0\\1&1&0&0\end{bmatrix},\ D=\begin{bmatrix}3&0&0&0\\0&3&0&0\\0&0&2&0\\0&0&0&2\end{bmatrix}$$

通过 `mapTriplets` 的计算，我们可以得到从点 `v1` 到 `v2,v3,v4` 的边的权重分别为 `1/3,1/3,1/3` ;从点 `v2` 到 `v1,v3,v4` 的权重分别为 `1/3,1/3,1/3` ;从点 `v3` 到 `v1,v2` 的权重分别为 `1/2,1/2` ;从点 `v4` 到 `v1,v2` 的权重分别为 `1/2,1/2` 。将这个图转换为矩阵的形式，可以得到如下矩阵 `W` 。

$$W=\begin{bmatrix}0&\frac{1}{3}&\frac{1}{3}&\frac{1}{3}\\\frac{1}{3}&0&\frac{1}{3}&\frac{1}{3}\\\frac{1}{2}&\frac{1}{2}&0&0\\\frac{1}{2}&\frac{1}{2}&0&0\end{bmatrix}=\begin{bmatrix}\frac{1}{3}&0&0&0\\0&\frac{1}{3}&0&0\\0&0&\frac{1}{2}&0\\0&0&0&\frac{1}{2}\end{bmatrix}\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&0\\1&1&0&0\end{bmatrix}=D^{-1}A$$

通过代码计算的结果和通过矩阵运算得到的结果一致。因此该代码实现了$W=D^{-1}A$ 。

- （2）初始化v^{0}

根据选择的初始化模式的不同，我们可以使用不同的方法初始化v^{0} 。一种方式是随机初始化，一种方式是度（ `degree` ）初始化，下面分别来介绍这两种方式。

- 随机初始化

```
def randomInit(g: Graph[Double, Double]): Graph[Double, Double]
= {
    //给每个顶点指定一个随机数
    val r = g.vertices.mapPartitionsWithIndex(
      (part, iter) => {
        val random = new XORShiftRandom(part)
        iter.map { case (id, _) =>
          (id, random.nextGaussian())
        }
      }, preservesPartitioning = true).cache()
    //所有顶点的随机值的绝对值之和
    val sum = r.values.map(math.abs).sum()
    //取平均值
    val v0 = r.mapValues(x => x / sum)
    GraphImpl.fromExistingRDDs(VertexRDD(v0), g.edges)
  }
```

- 度初始化

```
def initDegreeVector(g: Graph[Double, Double]): Graph[Double, D
ouble] = {
    //所有顶点的度之和
    val sum = g.vertices.values.sum()
    //取度的平均值
    val v0 = g.vertices.mapValues(_ / sum)
    GraphImpl.fromExistingRDDs(VertexRDD(v0), g.edges)
  }
```

通过初始化之后，我们获得了向量v^{0} 。它包含所有的顶点，但是顶点特征值发生了改变。随机初始化后，特征值为随机值；度初始化后，特征为度的平均值。

在这里，度初始化的向量我们称为“度向量”。度向量会给图中度大的节点分配更多的初始化权重，使其值可以更平均和快速的分布，从而更快的局部收敛。详细情况请参考文献【1】。

- （**3**）快速迭代求最终的**v**

```
for (iter <- 0 until maxIterations if math.abs(diffDelta) > tol)
 {
      val msgPrefix = s"Iteration $iter"
      // 计算w*vt
      val v = curG.aggregateMessages[Double](
        //相似度与目标点的度相乘
        sendMsg = ctx => ctx.sendToSrc(ctx.attr * ctx.dstAttr),
        mergeMsg = _ + _,
        TripletFields.Dst).cache()
      // 计算||Wvt||_1，即第二章公式中的c
      val norm = v.values.map(math.abs).sum()
      val v1 = v.mapValues(x => x / norm)
      // 计算v_t+1和v_t的不同
      val delta = curG.joinVertices(v1) { case (_, x, y) =>
        math.abs(x - y)
      }.vertices.values.sum()
      diffDelta = math.abs(delta - prevDelta)
      // 更新v
      curG = GraphImpl.fromExistingRDDs(VertexRDD(v1), g.edges)
      prevDelta = delta
    }
```

在上述代码中，我们通过 `aggregateMessages` 方法计算Wv^{t} 。我们仍然以第（1）步的举例来说明这个方法。假设我们以度来初始化v^{0} ， 在第一次迭代中，我们可以得到 `v1` （注意这里的 `v1` 是上面举例的顶点）的特征值为 `(1/3)*(3/10)+(1/3)*(1/5)+(1/3)*(1/5)=7/30` ， `v2` 的特征值为 `7/30` ， `v3` 的特征值为 `3/10` , `v4` 的特征值为 `3/10` 。即满足下面的公式。

$$v^1 = \begin{bmatrix} \frac{7}{30} \\ \frac{7}{30} \\ \frac{3}{10} \\ \frac{3}{10} \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{3}{10} \\ \frac{3}{10} \\ \frac{1}{5} \\ \frac{1}{5} \end{bmatrix} = Wv^0$$

- （**4**）使用 `k-means` 算法对**v**进行聚类

```
def kMeans(v: VertexRDD[Double], k: Int): VertexRDD[Int] = {
    val points = v.mapValues(x => Vectors.dense(x)).cache()
    val model = new KMeans()
      .setK(k)
      .setRuns(5)
      .setSeed(0L)
      .run(points.values)
    points.mapValues(p => model.predict(p)).cache()
  }
```

如果对 `graphX` 不太了解，可以阅读spark graph使用和源码解析

4 参考文献

【1】Frank Lin,William W. Cohen.Power Iteration Clustering

【2】漫谈 Clustering (4): Spectral Clustering

隐式狄利克雷分布

前言

`LDA` 是一种概率主题模型：隐式狄利克雷分布（ `Latent Dirichlet Allocation` ，简称 `LDA` ）。 `LDA` 是2003年提出的一种主题模型，它可以将文档集中每篇文档的主题以概率分布的形式给出。 通过分析一些文档，我们可以抽取出它们的主题（分布），根据主题（分布）进行主题聚类或文本分类。同时，它是一种典型的词袋模型，即一篇文档是由一组词构成，词与词之间没有先后顺序的关系。一篇文档可以包含多个主题，文档中每一个词都由其中的一个主题生成。

举一个简单的例子，比如假设事先给定了这几个主题： `Arts、Budgets、Children、Education` ，然后通过学习的方式，获取每个主题 `Topic` 对应的词语，如下图所示：

"Arts"	"Budgets"	"Children"	"Education"
NEW	MILLION	CHILDREN	SCHOOL
FILM	TAX	WOMEN	STUDENTS
SHOW	PROGRAM	PEOPLE	SCHOOLS
MUSIC	BUDGET	CHILD	EDUCATION
MOVIE	BILLION	YEARS	TEACHERS
PLAY	FEDERAL	FAMILIES	HIGH
MUSICAL	YEAR	WORK	PUBLIC
BEST	SPENDING	PARENTS	TEACHER
ACTOR	NEW	SAYS	BENNETT
FIRST	STATE	FAMILY	MANIGAT
YORK	PLAN	WELFARE	NAMPHY
OPERA	MONEY	MEN	STATE
THEATER	PROGRAMS	PERCENT	PRESIDENT
ACTRESS	GOVERNMENT	CARE	ELEMENTARY
LOVE	CONGRESS	LIFE	HAITI

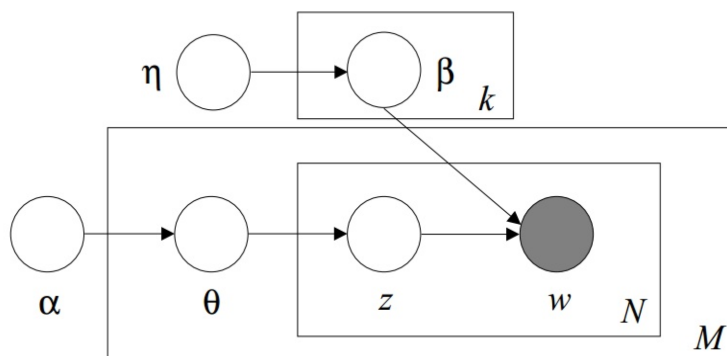
然后以一定的概率选取上述某个主题，再以一定的概率选取那个主题下的某个单词，不断的重复这两步，最终生成如下图所示的一篇文章（不同颜色的词语分别表示不同主题）。

The William Randolph Hearst Foundation will give $1.25 million to Lincoln Center, Metropolitan Opera Co., New York Philharmonic and Juilliard School. “Our board felt that we had a real opportunity to make a mark on the future of the performing arts with these grants an act every bit as important as our traditional areas of support in health, medical research, education and the social services,” Hearst Foundation President Randolph A. Hearst said Monday in announcing the grants. Lincoln Center’s share will be $200,000 for its new building, which will house young artists and provide new public facilities. The Metropolitan Opera Co. and New York Philharmonic will receive $400,000 each. The Juilliard School, where music and the performing arts are taught, will get $250,000. The Hearst Foundation, a leading supporter of the Lincoln Center Consolidated Corporate Fund, will make its usual annual $100,000 donation, too.

我们看到一篇文章后，往往会推测这篇文章是如何生成的，我们通常认为作者会先确定几个主题，然后围绕这几个主题遣词造句写成全文。`LDA` 要干的事情就是根据给定的文档，判断它的主题分布。在 `LDA` 模型中，生成文档的过程有如下几步：

- 从狄利克雷分布α中生成文档i的主题分布θ_{i} ；
- 从主题的多项式分布θ{i}中取样生成文档i第j个词的主题$Z{i,j}$ ；
- 从狄利克雷分布η中取样生成主题$Z{i,j}$对应的词语分布$\beta{i,j}$ ；
- 从词语的多项式分布$\beta{i,j}$中采样最终生成词语$W{i,j}$.

`LDA` 的图模型结构如下图所示：



`LDA` 会涉及很多数学知识，后面的章节我会首先介绍 `LDA` 涉及的数学知识，然后在这些数学知识的基础上详细讲解 `LDA` 的原理。

1 数学预备

1.1 Gamma函数

在高等数学中，有一个长相奇特的 `Gamma` 函数

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$

通过分部积分，可以推导 `gamma` 函数有如下递归性质

$$\Gamma(x+1) = x\Gamma(x)$$

通过该递归性质，我们可以很容易证明，`gamma` 函数可以被当成阶乘在实数集上的延拓，具有如下性质

$$\Gamma(n) = (n-1)!$$

1.2 Digamma函数

如下函数被称为 `Digamma` 函数，它是 `Gamma` 函数对数的一阶导数

$$\Psi(x) = \frac{d \log \Gamma(x)}{dx}$$

这是一个很重要的函数，在涉及 `Dirichlet` 分布相关的参数的极大似然估计时，往往需要用到这个函数。 `Digamma` 函数具有如下一个漂亮的性质

$$\Psi(x+1) = \Psi(x) + \frac{1}{x}$$

1.3 二项分布（Binomial distribution）

二项分布是由伯努利分布推出的。伯努利分布，又称两点分布或 `0-1` 分布，是一个离散型的随机分布，其中的随机变量只有两类取值，即0或者1。二项分布是重复 `n` 次的伯努利试验。简言之，只做一次实验，是伯努利分布，重复做了n次，是二项分布。二项分布的概率密度函数为：

$$P(K=k)=C(n,k)p^k(1-p)^{n-k}$$

对于k=1,2，...,n，其中 `C(n,k)` 是二项式系数（这就是二项分布的名称的由来）

$$C(n,k)=\frac{n!}{k!(n-k)!}$$

1.4 多项分布

多项分布是二项分布扩展到多维的情况。多项分布是指单次试验中的随机变量的取值不再是 `0-1` ，而是有多种离散值可能 `（1,2,3...,k）` 。比如投掷6个面的骰子实验， `N` 次实验结果服从 `K=6` 的多项分布。其中：

$$\sum_{i=1}^{k}p_i=1,p_i>1$$

多项分布的概率密度函数为：

$$P(x_1,x_2,\dots,x_k;n,p_1,p_2,\dots,p_k)=\frac{n!}{x_1!\dots x_k!}p_1^{x_1}\dots p_k^{x_k}$$

1.5 Beta分布

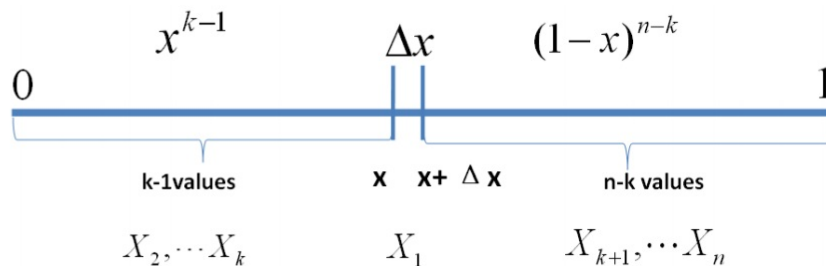
1.5.1 Beta分布

首先看下面的问题1（问题1到问题4都取自于文献【1】）。

问题**1**：

1: $X_1, X_2, \cdots, X_n \overset{\text{iid}}{\sim} Uniform(0,1)$,
2: 把这n 个随机变量排序后得到顺序统计量$X_{(1)}, X_{(2)} \cdots, X_{(n)}$,
3: 问$X_{(k)}$ 的分布是什么

为解决这个问题，可以尝试计算$x_{(k)}$落在区间 `[x,x+delta x]` 的概率。首先，把 `[0,1]` 区间分成三段 `[0,x)` ，`[x,x+delta x]` ，`(x+delta x,1]` ，然后考虑下简单的情形：即假设n个数中只有1个落在了区间 `[x,x+delta x]` 内，由于这个区间内的数 `X(k)` 是第k大的，所以 `[0,x)` 中应该有k-1个数， `(x+delta x,1]` 这个区间中应该有n-k个数。 如下图所示：



上述问题可以转换为下述事件 `E` ：

$$
\begin{aligned}
E = \{&X_1 \in [x, x+\Delta x], \\
&X_i \in [0, x) \quad (i = 2, \cdots, k), \\
&X_j \in (x+\Delta x, 1] \quad (j = k+1, \cdots, n)\}
\end{aligned}
$$

对于上述事件 `E` ，有：

$$
\begin{aligned}
P(E) &= \prod_{i=1}^{n} P(X_i) \\
&= x^{k-1}(1-x-\Delta x)^{n-k}\Delta x \\
&= x^{k-1}(1-x)^{n-k}\Delta x + o(\Delta x)
\end{aligned}
$$

其中， `o(delta x)` 表示 `delta x` 的高阶无穷小。显然，由于不同的排列组合，即n个数中有一个落在 `[x,x+delta x]` 区间的有n种取法，余下n-1个数中有k-1个落在 `[0,x)` 的有 `C(n-1,k-1)` 种组合。所以和事件E等价的事件一共有 `nC(n-1,k-1)` 个。

文献【1】中证明，只要落在 `[x,x+delta x]` 内的数字超过一个，则对应的事件的概率就是 `o(delta x)` 。所以$x_{(k)}$的概率密度函数为：

$$
\begin{aligned}
f(x) &= \lim_{\Delta x \to 0} \frac{P(x \le X_{(k)} \le x + \Delta x)}{\Delta x} \\
&= n \binom{n-1}{k-1} x^{k-1} (1-x)^{n-k} \\
&= \frac{n!}{(k-1)!(n-k)!} x^{k-1} (1-x)^{n-k} \quad x \in [0,1]
\end{aligned}
$$

利用 `Gamma` 函数，我们可以将 `f(x)` 表示成如下形式：

$$
f(x) = \frac{\Gamma(n+1)}{\Gamma(k)\Gamma(n-k+1)} x^{k-1} (1-x)^{n-k}
$$

在上式中，我们用 `alpha=k` ， `beta=n-k+1` 替换，可以得到 `beta` 分布的概率密度函数

$$
f(x) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1} (1-x)^{\beta-1}
$$

1.5.2 共轭先验分布

什么是共轭呢？轭的意思是束缚、控制。共轭从字面上理解，则是共同约束，或互相约束。在贝叶斯概率理论中，如果后验概率 `P(z|x)` 和先验概率 `p(z)` 满足同样的分布，那么，先验分布和后验分布被叫做共轭分布，同时，先验分布叫做似然函数的共轭先验分布。

1.5.3 Beta-Binomial 共轭

我们在问题1的基础上增加一些观测数据，变成问题**2**：

1: $X_1, X_2, \cdots, X_n \overset{\text{iid}}{\sim} Uniform(0,1)$，排序后对应的顺序统计量$X_{(1)}, X_{(2)} \cdots, X_{(n)}$，我们要猜测$p = X_{(k)}$；
2: $Y_1, Y_2, \cdots, Y_m \overset{\text{iid}}{\sim} Uniform(0,1)$，$Y_i$中有$m_1$个比$p$小，$m_2$个比$p$大；
3: 问$P(p|Y_1, Y_2, \cdots, Y_m)$ 的分布是什么。

第2步的条件可以用另外一句话来表述，即“ `Yi` 中有 `m1` 个比 `X(k`) 小， `m2` 个比 `X(k)` 大”，所以 `X(k)` 是 $X{(1)},X{(2)},...,X{(n)};Y{(1)},Y{(2)},...,Y{(m)}$中k+m1大的数。

根据1.5.1的介绍，我们知道事件p服从 `beta` 分布,它的概率密度函数为：

$$Beta(p|k+m_1,n-k+1+m_2)$$

按照贝叶斯推理的逻辑，把以上过程整理如下：

- 1、p是我们要猜测的参数，我们推导出 `p` 的分布为 `f(p)=Beta(p|k,n-k+1)` ,称为 `p` 的先验分布
- 2、根据 `Yi` 中有 `m1` 个比 `p` 小，有 `m2` 个比 `p` 大， `Yi` 相当是做了 `m` 次伯努利实验，所以 `m1` 服从二项分布 `B(m,p)`
- 3、在给定了来自数据提供 `(m1,m2)` 知识后，p的后验分布变为 `f(p|m1,m2)=Beta(p|k+m1,n-k+1+m2)`

贝叶斯估计的基本过程是：

先验分布 + 数据的知识 = 后验分布

以上贝叶斯分析过程的简单直观的表示就是：

Beta(p|k,n-k+1) + BinomCount(m1,m2) = Beta(p|k+m1,n-k+1+m2)

更一般的，对于非负实数alpha和beta，我们有如下关系

Beta(p|alpha,beta) + BinomCount(m1,m2) = Beta(p|alpha+m1,beta+m2)

针对于这种观测到的数据符合二项分布，参数的先验分布和后验分布都是 `Beta` 分布的情况，就是 `Beta-Binomial` 共轭。换言之， `Beta` 分布是二项式分布的共轭先验概率分布。二项分布和Beta分布是共轭分布意味着，如果我们为二项分布的参数p选取的先验分布是 `Beta` 分布，那么以p为参数的二项分布用贝叶斯估计得到的后验分布仍然服从 `Beta` 分布。

1.6 Dirichlet 分布

1.6.1 Dirichlet 分布

`Dirichlet` 分布，是 `beta` 分布在高维度上的推广。 `Dirichlet` 分布的的密度函数形式跟 `beta` 分布的密度函数类似：

$$P(x_1, x_2, \dots, x_k; \alpha_1, \alpha_2, \dots, \alpha_k) = \frac{1}{B(\alpha)} \prod_{i=1}^{k} x_i^{\alpha_k - 1}$$

其中

$$B(\alpha) = \frac{\prod_{i=1}^{k} \Gamma(\alpha_i)}{\Gamma(\sum_{i=1}^{k} \alpha_i)}, \sum x_i = 1$$

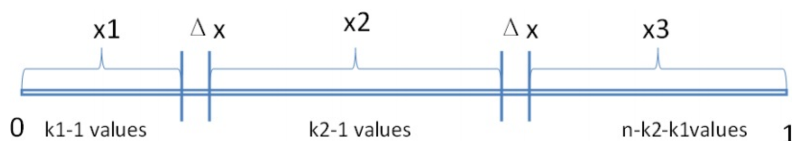
至此，我们可以看到二项分布和多项分布很相似，`Beta` 分布和 `Dirichlet` 分布很相似。并且 `Beta` 分布是二项式分布的共轭先验概率分布。那么 `Dirichlet` 分布呢？`Dirichlet` 分布是多项式分布的共轭先验概率分布。下文来论证这点。

1.6.2 Dirichlet-Multinomial 共轭

在1.5.3章问题2的基础上，我们更进一步引入问题**3**：

1: $X_1, X_2, \cdots, X_n \overset{\text{iid}}{\sim} Uniform(0,1)$,
2: 排序后对应的顺序统计量$X_{(1)}, X_{(2)} \cdots, X_{(n)}$,
3: 问$(X_{(k_1)}, X_{(k_1+k_2)})$的联合分布是什么;

类似于问题1的推导，我们可以容易推导联合分布。为了简化计算，我们取 `x3` 满足 `x1+x2+x3=1` , `x1` 和 `x2` 是变量。如下图所示。



概率计算如下：

$$\begin{aligned}
&P\Big(X_{(k_1)} \in (x_1, x_1 + \Delta x), X_{(k_1+k_2)} \in (x_2, x_2 + \Delta x)\Big) \\
&= n(n-1)\binom{n-2}{k_1 - 1, k_2 - 1} x_1^{k_1 - 1} x_2^{k_2 - 1} x_3^{n - k_1 - k_2} (\Delta x)^2 \\
&= \frac{n!}{(k_1 - 1)!(k_2 - 1)!(n - k_1 - k_2)!} x_1^{k_1 - 1} x_2^{k_2 - 1} x_3^{n - k_1 - k_2} (\Delta x)^2
\end{aligned}$$

于是我们得到联合分布为：

$$\begin{aligned} f(x_1, x_2, x_3) &= \frac{n!}{(k_1-1)!(k_2-1)!(n-k_1-k_2)!} x_1^{k_1-1} x_2^{k_2-1} x_3^{n-k_1-k_2} \\ &= \frac{\Gamma(n+1)}{\Gamma(k_1)\Gamma(k_2)\Gamma(n-k_1-k_2+1)} x_1^{k_1-1} x_2^{k_2-1} x_3^{n-k_1-k_2} \end{aligned}$$

观察上述式子的最终结果，可以看出上面这个分布其实就是3维形式的 `Dirichlet` 分布。令 `alpha1=k1,alpha2=k2,alpha3=n-k1-k2+1` ，分布密度函数可以写为：

$$f(x_1, x_2, x_3) = \frac{\Gamma(\alpha_1+\alpha_2+\alpha_3)}{\Gamma(\alpha_1)\Gamma(\alpha_2)\Gamma(\alpha_3)} x_1^{\alpha_1-1} x_2^{\alpha_2-1} x_3^{\alpha_3-1}$$

为了论证 `Dirichlet` 分布是多项式分布的共轭先验概率分布，在上述问题3的基础上再进一步，提出问题**4**。

1: $X_1, X_2, \cdots, X_n \overset{\text{iid}}{\sim} Uniform(0,1)$，排序后对应的顺序统计量$X_{(1)}, X_{(2)} \cdots, X_{(n)}$
2: 令$p_1 = X_{(k_1)}, p_2 = X_{(k_1+k_2)}, p_3 = 1 - p_1 - p_2$(加上$p_3$是为了数学表达简洁对称), 我们要猜测$\vec{p} = (p_1, p_2, p_3)$;
3: $Y_1, Y_2, \cdots, Y_m \overset{\text{iid}}{\sim} Uniform(0,1)$, Y_i中落到$[0, p_1), [p_1, p_2), [p_2, 1]$ 三个区间的个数分别为m_1, m_2, m_3，$m = m_1 + m_2 + m3$;
4: 问后验分布$P(\vec{p}|Y_1, Y_2, \cdots, Y_m)$ 的分布是什么。

为了方便计算，我们记

$$\vec{m} = (m_1, m_2, m_3), \quad \vec{k} = (k_1, k_2, n-k_1-k_2+1)$$

根据问题中的信息，我们可以推理得到 `p1,p2` 在 `X;Y` 这 `m+n` 个数中分别成为了第 `k1+m1,k1+k2+m1+m2` 大的数。后验分布p应该为

$$\mathrm{P}(\mathrm{p}|Y_1, Y_2, \ldots, Y_m) = \mathrm{Dir}(\mathrm{p}|k_1+m_1, k_2+m_2, n-k_1-k_2+1+m_3) = P(p|k+m)$$

同样的，按照贝叶斯推理的逻辑，可将上述过程整理如下：

- 1 我们要猜测参数 `P=(p1,p2,p3)` ，其先验分布为 `Dir(p|k)` ；
- 2 数据 `Yi` 落到三个区间 `[0,p1)` , `[p1,p2]` , `(p2,1]` 的个数分别是 `m1,m2,m3` ,所以 `m=(m1,m2,m3)` 服从多项分布 `Mult(m|p)` ；

- 3 在给定了来自数据提供的知识 `m` 后， `p` 的后验分布变为 `Dir(P|k+m)`

  上述贝叶斯分析过程的直观表述为：

  Dir(p|k) + Multcount(m) = Dir(p|k+m)

针对于这种观测到的数据符合多项分布，参数的先验分布和后验分布都是 `Dirichlet` 分布的情况，就是 `Dirichlet-Multinomial` 共轭。这意味着，如果我们为多项分布的参数p选取的先验分布是 `Dirichlet` 分布，那么以p为参数的多项分布用贝叶斯估计得到的后验分布仍然服从 `Dirichlet` 分布。

1.7 Beta和Dirichlet分布的一个性质

如果 `p=Beta(t|alpha,beta)` ，那么

$$
\begin{aligned}
E(p) &= \int_0^1 t * Beta(t|\alpha,\beta)dt \\
&= \int_0^1 t * \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} t^{\alpha-1}(1-t)^{\beta-1}dt \\
&= \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \int_0^1 t^{\alpha}(1-t)^{\beta-1}dt
\end{aligned}
$$

上式右边的积分对应到概率分布 `Beta(t|alpha+1,beta)` ，对于这个分布，我们有

$$
\int_0^1 \frac{\Gamma(\alpha+\beta+1)}{\Gamma(\alpha+1)\Gamma(\beta)} t^{\alpha}(1-t)^{\beta-1}dt = 1
$$

把上式带人 `E(p)` 的计算式，可以得到：

$$
\begin{aligned}
E(p) &= \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \cdot \frac{\Gamma(\alpha+1)\Gamma(\beta)}{\Gamma(\alpha+\beta+1)} \\
&= \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha+\beta+1)} \frac{\Gamma(\alpha+1)}{\Gamma(\alpha)} \\
&= \frac{\alpha}{\alpha+\beta}
\end{aligned}
$$

这说明，对于 `Beta` 分布的随机变量，其期望可以用上式来估计。 `Dirichlet` 分布也有类似的结论。对于 `p=Dir(t|alpha)` ，有

$$E(\vec{p}) = \left(\frac{\alpha_1}{\sum_{i=1}^{K} \alpha_i}, \frac{\alpha_2}{\sum_{i=1}^{K} \alpha_i}, \cdots, \frac{\alpha_K}{\sum_{i=1}^{K} \alpha_i} \right)$$

这个结论在后文的推导中会用到。

1.8 总结

`LDA` 涉及的数学知识较多，需要认真体会，以上大部分的知识来源于文献【1,2,3】,如有不清楚的地方，参见这些文献以了解更多。

2 主题模型LDA

在介绍LDA之前，我们先介绍几个基础模型：`Unigram model` 、 `mixture of unigrams model` 、 `pLSA model` 。为了方便描述，首先定义一些变量：

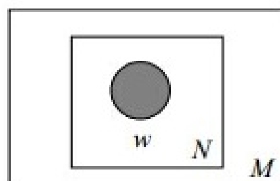
- 1 `w` 表示词， `V` 表示所有词的个数
- 2 `z` 表示主题， `k` 表示主题的个数
- 3 $D=(W{1},W{2},...,W_{M})$表示语料库， `M` 表示语料库中的文档数。
- 4 $W=(w{1},w{2},...,w_{N})$表示文档， `N` 表示文档中词的个数。

2.1 一元模型(Unigram model)

对于文档$W=(w{1},w{2},...,w{N})$,用$p(w{n})$表示w_{n}的先验概率，生成文档W的概率为：

$$P(W) = \prod_{n=1}^{N} p(w_n)$$

其图模型为（图中被涂色的 `w` 表示可观测变量， `N` 表示一篇文档中总共 `N` 个单词， `M` 表示 `M` 篇文档）：

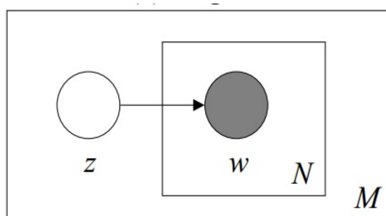


2.2 混合一元模型(Mixture of unigrams model)

该模型的生成过程是：给某个文档先选择一个主题 `Z` ，再根据该主题生成文档，该文档中的所有词都来自一个主题。生成文档的概率为：

$$p(\mathbf{w}) = \sum_z p(z) \prod_{n=1}^{N} p(w_n | z).$$

其图模型为（图中被涂色的 `w` 表示可观测变量，未被涂色的 `z` 表示未知的隐变量， `N` 表示一篇文档中总共 `N` 个单词， `M` 表示 `M` 篇文档）：



2.3 pLSA模型

在混合一元模型中，假定一篇文档只由一个主题生成，可实际中，一篇文章往往有多个主题，只是这多个主题各自在文档中出现的概率大小不一样。
在 `pLSA` 中，假设文档由多个主题生成。下面通过一个投色子的游戏（取自文献【2】的例子）说明 `pLSA` 生成文档的过程。

首先，假定你一共有 `K` 个可选的主题，有 `V` 个可选的词。假设你每写一篇文档会制作一颗 `K` 面的“文档-主题”骰子（扔此骰子能得到 `K` 个主题中的任意一个），和 `K` 个 `V` 面的“主题-词项”骰子（每个骰子对应一个主题， `K` 个骰子对应之前的 `K` 个主题，且骰子的每一面对应要选择的词项， `V` 个面对应着 `V` 个可选的词）。 比如可令 `K=3` ，即制作1个含有3个主题的“文档-主题”骰子，这3个主题可以是：教育、经济、交通。然后令 `V = 3` ，制作3个有着3面的“主题-词项”骰子，其中，教育主题骰子的3个面上的词可以是：大学、老师、课程，经济主题骰子的3个面上的词可以是：市场、企业、金融，交通主题骰子的3个面上的词可以是：高铁、汽车、飞机。

其次，每写一个词，先扔该“文档-主题”骰子选择主题，得到主题的结果后，使用和主题结果对应的那颗“主题-词项”骰子，扔该骰子选择要写的词。先扔“文档-主题”的骰子，假设以一定的概率得到的主题是：教育，所以下一步便是扔教育主题筛子，以一定的概率得到教育主题筛子对应的某个词大学。

- 上面这个投骰子产生词的过程简化一下便是：“先以一定的概率选取主题，再以一定的概率选取词”。事实上，一开始可供选择的主题有3个：教育、经济、交通，那为何偏偏选取教育这个主题呢？其实是随机选取的，只是这个随机遵循一定的概率分布。比如可能选取教育主题的概率是0.5，选取经济主题的概率是0.3，选取交通主题的概率是0.2，那么这3个主题的概率分布便是 `{教育：0.5，经济：0.3，交通：0.2}` ，我们把各个主题 `z` 在文档 `d` 中出现的概率分布称之为主题分布，且是一个多项分布。

- 同样的，从主题分布中随机抽取出教育主题后，依然面对着3个词：大学、老师、课程，这3个词都可能被选中，但它们被选中的概率也是不一样的。比如大学这个词被选中的概率是0.5，老师这个词被选中的概率是0.3，课程被选中的概率是0.2，那么这3个词的概率分布便是 `{大学：0.5，老师：0.3，课程：0.2}` ，我们把各个词语w在主题z下出现的概率分布称之为词分布，这个词分布也是一个多项分布。

- 所以，选主题和选词都是两个随机的过程，先从主题分布 `{教育：0.5，经济：0.3，交通：0.2}` 中抽取出主题：教育，然后从该主题对应的词分布 `{大学：0.5，老师：0.3，课程：0.2}` 中抽取出词：大学。

最后，你不停的重复扔“文档-主题”骰子和”主题-词项“骰子，重复 `N` 次（产生 `N` 个词），完成一篇文档，重复这产生一篇文档的方法 `M` 次，则完成 `M` 篇文档。

上述过程抽象出来即是 `pLSA` 的文档生成模型。在这个过程中，我们并未关注词和词之间的出现顺序，所以 `pLSA` 是一种词袋模型。定义如下变量:

- $(z_{1},z_{2},...,z_{k})$表示隐藏的主题；

- $P(d_{i})$表示海量文档中某篇文档被选中的概率；

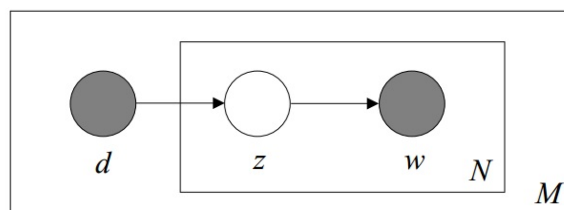
- $P(w_{j}|d_{i})$表示词w_{j}在文档d_{i}中出现的概率；针对海量文档，对所有文档进行分词后，得到一个词汇列表，这样每篇文档就是一个词语的集合。对于每个词语，用它在文档中出现的次数除以文档中词语总的数目便是它在文档中出现的概率；

- $P(z_{k}|d_{i})$表示主题z_{k}在文档d_{i}中出现的概率；
- $P(w_{j}|z_{k})$表示词w_{j}在主题z_{k}中出现的概率。与主题关系越密切的词其条件概率越大。

我们可以按照如下的步骤得到“文档-词项”的生成模型：

- 1 按照$P(d_{i})$选择一篇文档d_{i} ；
- 2 选定文档d_{i}之后，从主题分布中按照概率$P(z_{k}|d_{i})$选择主题；
- 3 选定主题后，从词分布中按照概率$P(w_{j}|z_{k})$选择一个词。

利用看到的文档推断其隐藏的主题（分布）的过程，就是主题建模的目的：自动地发现文档集中的主题（分布）。文档 `d` 和单词 `w` 是可被观察到的，但主题 `z` 却是隐藏的。如下图所示（图中被涂色的 `d`、`w` 表示可观测变量，未被涂色的 `z` 表示未知的隐变量，`N` 表示一篇文档中总共 `N` 个单词，`M` 表示 `M` 篇文档）。



上图中，文档 `d` 和词 `w` 是我们得到的样本，可观测得到，所以对于任意一篇文档，其$P(w_{j}|d_{i})$是已知的。根据这个概率可以训练得到 `文档-主题` 概率以及 `主题-词项` 概率。即：

$$\mathrm{P}(w_j|d_i) = \sum_{k=1}^{K} P(w_j|z_k)P(z_k|d_i)$$

故得到文档中每个词的生成概率为：

$$\mathrm{P}(w_j, d_i) = \mathrm{P}(d_i)\mathrm{P}(w_j|d_i) = \mathrm{P}(d_i)\sum_{k=1}^{K} P(w_j|z_k)P(z_k|d_i)$$

$P(d{i})$可以直接得出，而$P(z{k}|d{i})$和$P(w{j}|z{k})$未知，所以 $\theta=(P(z{k}|d{i}),P(w{j}|z_{k}))$就是我们要估计的参数,我们要最大化这个参数。因为该待估计的参数中含有隐变量 `z` ，所以我们可以用 `EM` 算法来估计这个参数。

2.4 LDA模型

`LDA` 的不同之处在于， `pLSA` 的主题的概率分布 `P(c|d)` 是一个确定的概率分布，也就是虽然主题 `c` 不确定，但是 `c` 符合的概率分布是确定的，比如符合某个多项分布，这个多项分布的各参数是确定的。 但是在 `LDA` 中，这个多项分布都是不确定的，高斯分布又服从一个狄利克雷先验分布 `(Dirichlet prior)` 。即 `LDA` 就是 `pLSA` 的贝叶斯版本,正因为 `LDA` 被贝叶斯化了，所以才会加的两个先验参数。

`LDA` 模型中一篇文档生成的方式如下所示:

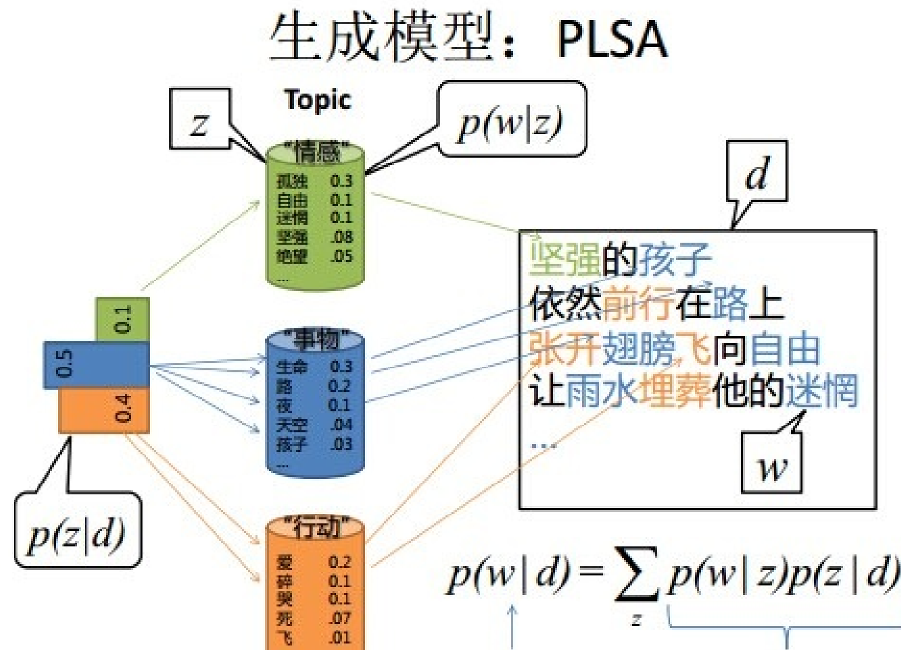
- 1 按照$P(d{i})$选择一篇文档$d{i}$ ；
- 2 从狄利克雷分布α中生成文档i的主题分布θ_{i} ；
- 3 从主题的多项式分布θ{i}中取样生成文档i第j个词的主题$Z{i,j}$ ；
- 4 从狄利克雷分布η中取样生成主题$Z{i,j}$对应的词语分布$\beta{i,j}$ ；
- 5 从词语的多项式分布$\beta{i,j}$中采样最终生成词语$W{i,j}$

从上面的过程可以看出， `LDA` 在 `pLSA` 的基础上，为主题分布和词分布分别加了两个 `Dirichlet` 先验。

拿之前讲解 `pLSA` 的例子进行具体说明。如前所述，在 `pLSA` 中，选主题和选词都是两个随机的过程，先从主题分布 `{教育：0.5，经济：0.3，交通：0.2}` 中抽取出主题：教育，然后从该主题对应的词分布 `{大学：0.5，老师：0.3，课程：0.2}` 中抽取出词：大学。 在 `LDA` 中，选主题和选词依然都是两个随机的过程。但在 `LDA` 中，主题分布和词分布不再唯一确定不变，即无法确切给出。例如主题分布可能是 `{教育：0.5，经济：0.3，交通：0.2}` ，也可能是 `{教育：0.6，经济：0.2，交通：0.2}` ，到底是哪个我们不能确定，因为它是随机的可变化的。 但再怎么变化，也依然服从一定的分布，主题分布和词分布由 `Dirichlet` 先验确定。

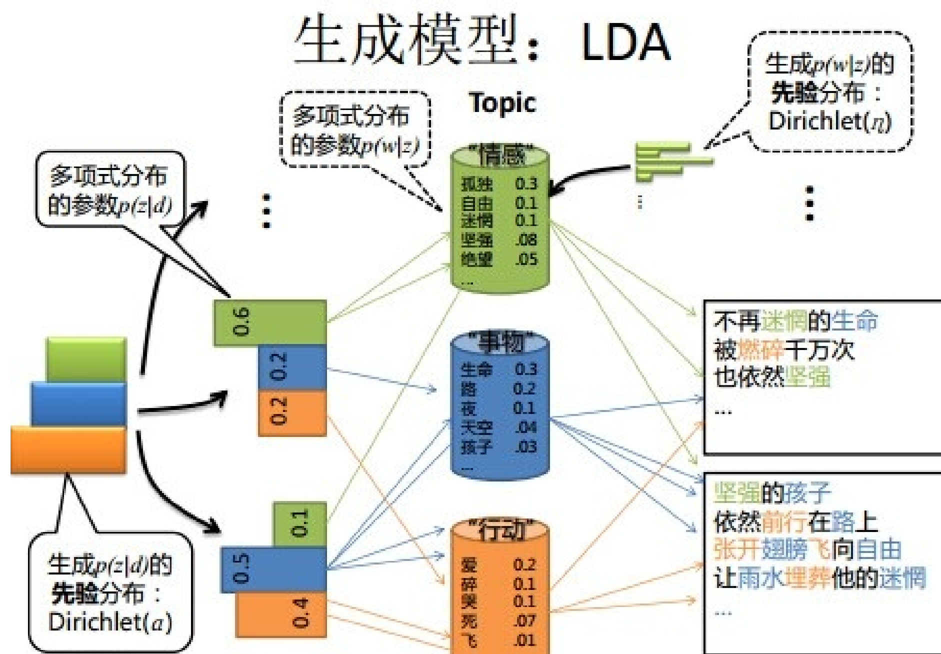
举个文档 `d` 产生主题 `z` 的例子。

在 pLSA 中，给定一篇文档d，主题分布是一定的，比如 { P(zi|d), i = 1,2,3 } 可能就是 {0.4,0.5,0.1} ，表示 z1、z2、z3 这3个主题被文档 d 选中的概率都是个固定的值： P(z1|d) = 0.4、P(z2|d) = 0.5、P(z3|d) = 0.1 ，如下图所示:

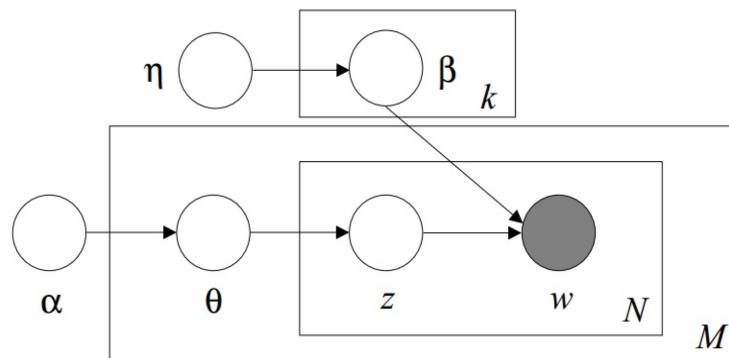


在 LDA 中，主题分布（各个主题在文档中出现的概率分布）和词分布（各个词语在某个主题下出现的概率分布）是唯一确定的。 LDA 为提供了两个 Dirichlet 先验参数， Dirichlet 先验为某篇文档随机抽取出主题分布和词分布。

给定一篇文档 d ，现在有多个主题 z1、z2、z3 ，它们的主题分布 { P(zi|d), i = 1,2,3 } 可能是 {0.4,0.5,0.1} ，也可能是 {0.2,0.2,0.6} ，即这些主题被 d 选中的概率都不再是确定的值，可能是 P(z1|d) = 0.4、P(z2|d) = 0.5、P(z3|d) = 0.1 ，也有可能是 P(z1|d) = 0.2、P(z2|d) = 0.2、P(z3|d) = 0.6 ，而主题分布到底是哪个取值集合我们不确定，但其先验分布是 dirichlet 分布，所以可以从无穷多个主题分布中按照 dirichlet 先验随机抽取出某个主题分布出来。如下图所示



LDA 在 pLSA 的基础上给两参数$P(z_{k}|d_{i})$和$P(w_{j}|z_{k})$加了两个先验分布的参数。这两个分布都是 Dirichlet 分布。 下面是 LDA 的图模型结构：



3 LDA 参数估计

在 spark 中，提供了两种方法来估计参数，分别是变分 EM （期望最大）算法（见文献【3】【4】）和在线学习算法（见文献【5】）。下面将分别介绍这两种算法以及其源码实现。

3.1 变分EM算法

变分贝叶斯算法的详细信息可以参考文献【9】。

在上文中，我们知道 `LDA` 将变量 `theta` 和 `phi` （为了方便起见，我们将上文 `LDA` 图模型中的 `beta` 改为了 `phi` ）看做随机变量，并且为 `theta` 添加一个超参数为 `alpha` 的 `Dirichlet` 先验，为 `phi` 添加一个超参数为 `eta` 的 `Dirichlet` 先验来估计 `theta` 和 `beta` 的最大后验（ `MAP` ）。可以通过最优化最大后验估计来估计参数。我们首先来定义几个变量：

- 下式的 `gamma` 表示词为 `w` ，文档为 `j` 时，主题为 `k` 的概率，如公式（**3.1.1**）

$$\gamma_{wjk} = P(z = k|x = w, d = j)$$

- N_{wj}表示词 `w` 在文档 `j` 中出现的次数；
- N_{wk}表示词 `w` 在主题 `k` 中出现的次数，如公式（**3.1.2**）

$$N_{wk} = \sum_j N_{wj}\gamma_{wjk}$$

- N_{kj}表示主题 `k` 在文档 `j` 中出现的次数，如公式（**3.1.3**）

$$N_{kj} = \sum_w N_{wj}\gamma_{wjk}$$

- N_{k}表示主题 `k` 中包含的词出现的总次数，如公式（**3.1.4**）

$$N_k = \sum_w N_{wk}$$

- N_{j}表示文档 `j` 中包含的主题出现的总次数,如公式（**3.1.5**）

$$N_j = \sum_k N_{kj}$$

根据文献【4】中 `2.2` 章节的介绍，我们可以推导出如下更新公式**(3.1.6)**，其中 `alpha` 和 `eta` 均大于1：

$$\gamma_{wjk} \propto \frac{(N_{wk}+\eta-1)(N_{kj}+\alpha-1)}{(N_k+W\eta-W)}$$

收敛之后，最大后验估计可以得到公式**(3.1.7)**：

$$\hat{\phi}_{wk} = \frac{N_{wk}+\eta-1}{N_k+W\eta-W} \quad \hat{\theta}_{kj} = \frac{N_{kj}+\alpha-1}{N_j+K\alpha-K}$$

变分 `EM` 算法的流程如下：

- **1** 初始化状态，即随机初始化$N*{wk}$和$N*{kj}$
- **2 E-**步，对每一个 `（文档，词汇）` 对 `i` ，计算$P(z*{i}|w*{i},d_{i})$，更新 `gamma` 值
- **3 M-**步，计算隐藏变量 `phi` 和 `theta` 。即计算$N*{wk}$和$N*{kj}$
- **4** 重复以上**2**、**3**两步，直到满足最大迭代数

第 `4.2` 章会从代码层面说明该算法的实现流程。

3.2 在线学习算法

3.2.1 批量变分贝叶斯

在变分贝叶斯推导(`VB`)中，根据文献【3】，使用一种更简单的分布 `q(z,theta,beta)` 来估计真正的后验分布，这个简单的分布使用一组自由变量(`free parameters`)来定义。通过最大化对数似然的一个下界（ `Evidence Lower Bound (ELBO)` ）来最优化这些参数，如下公式**(3.2.1)**

$$\log p(\boldsymbol{w}|\alpha,\eta) \geq \mathcal{L}(\boldsymbol{w},\boldsymbol{\phi},\boldsymbol{\gamma},\boldsymbol{\lambda}) \triangleq \mathbb{E}_q[\log p(\boldsymbol{w},\boldsymbol{z},\boldsymbol{\theta},\boldsymbol{\beta}|\alpha,\eta)] - \mathbb{E}_q[\log q(\boldsymbol{z},\boldsymbol{\theta},\boldsymbol{\beta})]$$

最大化 `ELBO` 就是最小化 `q(z,theta,beta)` 和 `p(z,theta,beta|w,alpha,eta)` 的 `KL` 距离。根据文献【3】，我们将 `q` 因式分解为如下（**3.2.2**）的形式：

$$q(z_{di}=k)=\phi_{dw_{di}k}; \quad q(\theta_d)=\text{Dirichlet}(\theta_d;\gamma_d); \quad q(\beta_k)=\text{Dirichlet}(\beta_k;\lambda_k)$$

后验 `z` 通过 `phi` 来参数化，后验 `theta` 通过 `gamma` 来参数化，后验 `beta` 通过 `lambda` 来参数化。为了简单描述，我们把 `lambda` 当作“主题”来看待。公式**(3.2.2)**分解为如下**(3.2.3)**形式：

$$\mathcal{L}(\boldsymbol{w},\boldsymbol{\phi},\boldsymbol{\gamma},\boldsymbol{\lambda}) = \sum_d \{\mathbb{E}_q[\log p(w_d|\theta_d,z_d,\boldsymbol{\beta})] + \mathbb{E}_q[\log p(z_d|\theta_d)] - \mathbb{E}_q[\log q(z_d)] \\ + \mathbb{E}_q[\log p(\theta_d|\alpha)] - \mathbb{E}_q[\log q(\theta_d)] + (\mathbb{E}_q[\log p(\boldsymbol{\beta}|\eta)] - \mathbb{E}_q[\log q(\boldsymbol{\beta})])/D\}$$

我们现在将上面的期望扩展为变分参数的函数形式。这反映了变分目标只依赖于n_{dw} ，即词 `w` 出现在文档 `d` 中的次数。当使用 `VB` 算法时，文档可以通过它们的词频来汇总（ `summarized` ），如公式**(3.2.4)**

$$\begin{aligned}\mathcal{L} &= \sum_d \sum_w n_{dw} \sum_k \phi_{dwk}(\mathbb{E}_q[\log\theta_{dk}] + \mathbb{E}_q[\log\beta_{kw}] - \log\phi_{dwk}) \\ &\quad - \log\Gamma(\textstyle\sum_k \gamma_{dk}) + \sum_k(\alpha - \gamma_{dk})\mathbb{E}_q[\log\theta_{dk}] + \log\Gamma(\gamma_{dk}) \\ &\quad + (\textstyle\sum_k - \log\Gamma(\sum_w \lambda_{kw}) + \sum_w(\eta - \lambda_{kw})\mathbb{E}_q[\log\beta_{kw}] + \log\Gamma(\lambda_{kw}))/D \\ &\quad + \log\Gamma(K\alpha) - K\log\Gamma(\alpha) + (\log\Gamma(W\eta) - W\log\Gamma(\eta))/D \\ &\triangleq \textstyle\sum_d \ell(n_d,\phi_d,\gamma_d,\boldsymbol{\lambda}),\end{aligned}$$

上面的公式中， `W` 表示词的数量， `D` 表示文档的数量。 `l` 表示文档 `d` 对 `ELBO` 所做的贡献。 `L` 可以通过坐标上升法来最优化，它的更新公式如**(3.2.5)**:

$$\phi_{dwk} \propto \exp\{\mathbb{E}_q[\log\theta_{dk}] + \mathbb{E}_q[\log\beta_{kw}]\}; \quad \gamma_{dk} = \alpha + \sum_w n_{dw}\phi_{dwk}; \quad \lambda_{kw} = \eta + \sum_d n_{dw}\phi_{dwk}$$

`log(theta)` 和 `log(beta)` 的期望通过下面的公式**(3.2.6)**计算：

$$\mathbb{E}_q[\log\theta_{dk}] = \Psi(\gamma_{dk}) - \Psi(\textstyle\sum_{i=1}^K \gamma_{di}); \quad \mathbb{E}_q[\log\beta_{kw}] = \Psi(\lambda_{kw}) - \Psi(\sum_{i=1}^W \lambda_{ki}).$$

通过 `EM` 算法，我们可以将这些更新分解成 `E-步` 和 `M-步` 。 `E-步` 固定 `lambda` 来更新 `gamma` 和 `phi` ； `M-步` 通过给定 `phi` 来更新 `lambda` 。批 `VB` 算法的过程如下**(**算法**1)**所示：

Initialize $\boldsymbol{\lambda}$ randomly.
while relative improvement in $\mathcal{L}(\boldsymbol{w}, \boldsymbol{\phi}, \boldsymbol{\gamma}, \boldsymbol{\lambda}) > 0.00001$ **do**
  E step:
  for $d = 1$ to D **do**
    Initialize $\gamma_{dk} = 1$. (The constant 1 is arbitrary.)
    repeat
      Set $\phi_{dwk} \propto \exp\{\mathbb{E}_q[\log \theta_{dk}] + \mathbb{E}_q[\log \beta_{kw}]\}$
      Set $\gamma_{dk} = \alpha + \sum_w \phi_{dwk} n_{dw}$
    until $\frac{1}{K}\sum_k |\text{change in} \gamma_{dk}| < 0.00001$
  end for
  M step:
  Set $\lambda_{kw} = \eta + \sum_d n_{dw}\phi_{dwk}$
end while

3.2.2 在线变分贝叶斯

批量变分贝叶斯算法需要固定的内存，并且比吉布斯采样更快。但是它仍然需要在每次迭代时处理所有的文档，这在处理大规模文档时，速度会很慢，并且也不适合流式数据的处理。 文献【5】提出了一种在线变分推导算法。设定 `gamma(n_d,lambda)` 和 `phi(n_d,lambda)` 分别表示 `gamma_d` 和 `phi_d` 的值，我们的目的就是设定 `phi` 来最大化下面的公式**(3.2.7)**

$$\mathcal{L}(\boldsymbol{n}, \boldsymbol{\lambda}) \triangleq \sum_d \ell(n_d, \gamma(n_d, \boldsymbol{\lambda}), \phi(n_d, \boldsymbol{\lambda}), \boldsymbol{\lambda})$$

我们在算法**2**中介绍了在线 `VB` 算法。因为词频的第t个向量n_{t}是可观察的，我们在 `E-步` 通过固定 `lambda` 来找到 `gamma_t` 和 `phi_t` 的局部最优解。然后，我们计算 `lambda_cap` 。如果整个语料库由单个文档重复 `D` 次组成，那么这样的 `lambda_cap` 设置是最优的。之后，我们通过 `lambda` 之前的值以及 `lambda_cap` 来更新 `lambda` 。我们给 `lambda_cap` 设置的权重如公式**(3.2.8)**所示：

$$\rho_t \triangleq (\tau_0 + t)^{-\kappa}, \kappa \in (0.5, 1]$$

在线 `VB` 算法的实现流程如下算法**2**所示

Define $\rho_t \triangleq (\tau_0 + t)^{-\kappa}$
Initialize $\boldsymbol{\lambda}$ randomly.
for $t = 0$ to ∞ **do**
  E step:
  Initialize $\gamma_{tk} = 1$. (The constant 1 is arbitrary.)
  repeat
    Set $\phi_{twk} \propto \exp\{\mathbb{E}_q[\log \theta_{tk}] + \mathbb{E}_q[\log \beta_{kw}]\}$
    Set $\gamma_{tk} = \alpha + \sum_w \phi_{twk} n_{tw}$
  until $\frac{1}{K}\sum_k |\text{change in} \gamma_{tk}| < 0.00001$
  M step:
  Compute $\tilde{\lambda}_{kw} = \eta + D n_{tw} \phi_{twk}$
  Set $\boldsymbol{\lambda} = (1-\rho_t)\boldsymbol{\lambda} + \rho_t \tilde{\boldsymbol{\lambda}}$.
end for

那么在在线`VB`算法中，`alpha`和`eta`是如何更新的呢？参考文献【8】提供了计算方法。给定数据集，`dirichlet`参数的可以通过最大化下面的对数似然来估计

$$
\begin{aligned}
F(\alpha) = \log p(D|\alpha) &= \log \prod_i p(\mathbf{p}_i|\alpha) \\
&= \log \prod_i \frac{\Gamma\left(\sum_k \alpha_k\right)}{\prod_k \Gamma(\alpha_k)} \prod_k p_{ik}^{\alpha_k - 1} \\
&= N\left(\log \Gamma\left(\sum_k \alpha_k\right) - \sum_k \log \Gamma(\alpha_k) + \sum_k (\alpha_k - 1) \log \hat{p}_k\right)
\end{aligned}
$$

其中，

$$\hat{p}_k = \frac{1}{N}\sum_i \log p_{ik}$$

有多种方法可以最大化这个目标函数，如梯度上升，`Newton-Raphson`等。`Spark`使用`Newton-Raphson`方法估计参数，更新`alpha`。`Newton-Raphson`提供了一种参数二次收敛的方法，它一般的更新规则如下公式**(3.3.3)**:

$$\alpha^{new} = \alpha^{old} - H^{-1}(F) \cdot \nabla F$$

其中，`H`表示海森矩阵。对于这个特别的对数似然函数，可以应用`Newton-Raphson`去解决高维数据，因为它可以在线性时间求出海森矩阵的逆矩阵。一般情况下，海森矩阵可以用一个对角矩阵和一个元素都一样的矩阵的和来表示。 如下公

式**(3.3.4)**，`Q` 是对角矩阵，`C11` 是元素相同的一个矩阵。

$$\begin{aligned} H &= Q + c11^T \\ q_{jk} &= -N\Psi'(\alpha_k)\delta(j-k) \\ c &= N\Psi'\left(\sum_k \alpha_k\right) \end{aligned}$$

为了计算海森矩阵的逆矩阵，我们观察到，对任意的可逆矩阵 `Q` 和非负标量 `c` ，有下列式子**(3.3.5)**:

$$\begin{aligned} (Q+c11^T)\left(Q^{-1}-\frac{Q^{-1}11^TQ^{-1}}{1/c+1^TQ^{-1}1}\right) &= QQ^{-1}-\frac{QQ^{-1}11^TQ^{-1}}{1/c+1^TQ^{-1}1}+c11^TQ^{-1} \\ &\quad -\frac{c11^TQ^{-1}11^TQ^{-1}}{1/c+1^TQ^{-1}1} \\ &= QQ^{-1}+\frac{1}{1/c+1^TQ^{-1}1}(-11^TQ^{-1}+11^TQ^{-1} \\ &\quad +c11^TQ^{-1}(1^TQ^{-1}1)-c1(1^TQ^{-1}1)1^TQ^{-1}) \\ &= 1 \end{aligned}$$

因为 `Q` 是对角矩阵，所以 `Q` 的逆矩阵可以很容易的计算出来。所以 `Newton-Raphson` 的更新规则可以重写为如下**(3.3.6)**的形式

$$\alpha_k^{new} = \alpha_k^{old} - \frac{(\nabla F)_k - b}{q_{kk}}$$

其中 `b` 如下公式**(3.3.7)**，

$$b = \frac{Q^{-1}11^TQ^{-1}}{1/c+1^TQ^{-1}1} = \frac{\sum_j (\nabla F)_j / q_{jj}}{1/z+\sum_j 1/q_{jj}}$$

4 LDA代码实现

4.1 LDA使用实例

我们从官方文档【6】给出的使用代码为起始点来详细分析 `LDA` 的实现。

```
import org.apache.spark.mllib.clustering.{LDA, DistributedLDAMod
el}
import org.apache.spark.mllib.linalg.Vectors
// 加载和处理数据
val data = sc.textFile("data/mllib/sample_lda_data.txt")
val parsedData = data.map(s => Vectors.dense(s.trim.split(' ').m
ap(_.toDouble)))
// 为文档编号，编号唯一。List（（id，vector）....）
val corpus = parsedData.zipWithIndex.map(_.swap).cache()
// 将文档聚类为3类
val ldaModel = new LDA().setK(3).run(corpus)
val topics = ldaModel.topicsMatrix
for (topic <- Range(0, 3)) {
  print("Topic " + topic + ":")
  for (word <- Range(0, ldaModel.vocabSize)) { print(" " + topic
s(word, topic)); }
  println()
}
```

以上代码主要做了两件事：加载和切分数据、训练模型。在样本数据中，每一行代表一篇文档，经过处理后， `corpus` 的类型为 `List((id,vector)*)` ，一个 `(id,vector)` 代表一篇文档。将处理后的数据传给 `org.apache.spark.mllib.clustering.LDA` 类的 `run` 方法， 就可以开始训练模型。 `run` 方法的代码如下所示：

```
def run(documents: RDD[(Long, Vector)]): LDAModel = {
    val state = ldaOptimizer.initialize(documents, this)
    var iter = 0
    val iterationTimes = Array.fill[Double](maxIterations)(0)
    while (iter < maxIterations) {
      val start = System.nanoTime()
      state.next()
      val elapsedSeconds = (System.nanoTime() - start) / 1e9
      iterationTimes(iter) = elapsedSeconds
      iter += 1
    }
    state.getLDAModel(iterationTimes)
  }
```

这段代码首先调用 `initialize` 方法初始化状态信息，然后循环迭代调用 `next` 方法直到满足最大的迭代次数。在我们没有指定的情况下，迭代次数默认为20。需要注意的是， `ldaOptimizer` 有两个具体的实现类 `EMLDAOptimizer` 和 `OnlineLDAOptimizer` ，它们分别表示使用 `EM` 算法和在线学习算法实现参数估计。在未指定的情况下，默认使用 `EMLDAOptimizer` 。

4.2 变分EM算法的实现

在 `spark` 中，使用 `GraphX` 来实现 `EMLDAOptimizer` ，这个图是有两种类型的顶点的二分图。这两类顶点分别是文档顶点（ `Document vertices` ）和词顶点（ `Term vertices` ）。

- 文档顶点使用大于0的唯一的指标来索引，保存长度为 `k` （主题个数）的向量
- 词顶点使用 `{-1, -2, ..., -vocabSize}` 来索引，保存长度为 `k` （主题个数）的向量
- 边（ `edges` ）对应词出现在文档中的情况。边的方向是 `document -> term` ，并且根据 `document` 进行分区

我们可以根据3.1节中介绍的算法流程来解析源代码。

4.2.1 初始化状态

`spark` 在 `EMLDAOptimizer` 的 `initialize` 方法中实现初始化功能。包括初始化 `Dirichlet` 参数 `alpha` 和 `eta` 、初始化边、初始化顶点以及初始化图。

```
//对应超参数alpha
val docConcentration = lda.getDocConcentration
//对应超参数eta
val topicConcentration = lda.getTopicConcentration
this.docConcentration = if (docConcentration == -1) (50.0 / k)
+ 1.0 else docConcentration
this.topicConcentration = if (topicConcentration == -1) 1.1 else
topicConcentration
```

上面的代码初始化了超参数 `alpha` 和 `eta` ，根据文献【4】，当 `alpha` 未指定时，初始化其为 `(50.0 / k) + 1.0` ，其中 `k` 表示主题个数。当 `eta` 未指定时，初始化其为1.1。

```
//对于每个文档，为每一个唯一的Term创建一个(Document->Term)的边
val edges: RDD[Edge[TokenCount]] = docs.flatMap { case (docID: Long, termCounts: Vector) =>
      // Add edges for terms with non-zero counts.
      termCounts.toBreeze.activeIterator.filter(_._2 != 0.0).map { case (term, cnt) =>
        //文档id，termindex，词频
        Edge(docID, term2index(term), cnt)
      }
}
//term2index将term转为{-1, -2, ..., -vocabSize}索引
 private[clustering] def term2index(term: Int): Long = -(1 + term.toLong)
```

上面的这段代码处理每个文档，对文档中每个唯一的 `Term` （词）创建一个边，边的格式为 `(文档id，词索引，词频)` 。词索引为 `{-1, -2, ..., -vocabSize}` 。

```
//创建顶点
 val docTermVertices: RDD[(VertexId, TopicCounts)] = {
      val verticesTMP: RDD[(VertexId, TopicCounts)] =
        edges.mapPartitionsWithIndex { case (partIndex, partEdge
s) =>
          val random = new Random(partIndex + randomSeed)
          partEdges.flatMap { edge =>
            val gamma = normalize(BDV.fill[Double](k)(random.nex
tDouble()), 1.0)
            //此处的sum是DenseVector，gamma*N_wj
            val sum = gamma * edge.attr
            //srcId表示文献id，dstId表是词索引
            Seq((edge.srcId, sum), (edge.dstId, sum))
          }
        }
      verticesTMP.reduceByKey(_ + _)
 }
```

上面的代码创建顶点。我们为每个主题随机初始化一个值，即 `gamma` 是随机的。 `sum` 为 `gamma * edge.attr` ，这里的 `edge.attr` 即 `N_wj` ,所以 `sum` 用 `gamma * N_wj` 作为顶点的初始值。

```
this.graph = Graph(docTermVertices, edges).partitionBy(Partition
Strategy.EdgePartition1D)
```

上面的代码初始化 `Graph` 并通过文档分区。

4.2.2 E-步：更新gamma

```
    val eta = topicConcentration
    val W = vocabSize
    val alpha = docConcentration
    val N_k = globalTopicTotals
    val sendMsg: EdgeContext[TopicCounts, TokenCount, (Boolean,
TopicCounts)] => Unit =
      (edgeContext) => {
        // 计算 N_{wj} gamma_{wjk}
        val N_wj = edgeContext.attr
        // E-STEP: 计算 gamma_{wjk} 通过N_{wj}来计算
        //此处的edgeContext.srcAttr为当前迭代的N_kj , edgeContext.d
stAttr为当前迭代的N_wk,
        //后面通过M-步，会更新这两个值,作为下一次迭代的当前值
        val scaledTopicDistribution: TopicCounts =
                    computePTopic(edgeContext.srcAttr, edgeConte
xt.dstAttr, N_k, W, eta, alpha) *= N_wj
        edgeContext.sendToDst((false, scaledTopicDistribution))
        edgeContext.sendToSrc((false, scaledTopicDistribution))
      }
```

上述代码中，`W` 表示词数，`N_k` 表示所有文档中，出现在主题 `k` 中的词的词频总数，后续的实现会使用方法 `computeGlobalTopicTotals` 来更新这个值。`N_wj` 表示词 `w` 出现在文档 `j` 中的词频数，为已知数。`E-步` 就是利用公式**(3.1.6)**去更新 `gamma` 。代码中使用 `computePTopic` 方法来实现这个更新。`edgeContext` 通过方法 `sendToDst` 将 `scaledTopicDistribution` 发送到目标顶点，通过方法 `sendToSrc` 发送到源顶点以便于后续的 `M-步` 更新的 `N_kj` 和 `N_wk` 。下面我们看看 `computePTopic` 方法。

```
private[clustering] def computePTopic(
      docTopicCounts: TopicCounts,
      termTopicCounts: TopicCounts,
      totalTopicCounts: TopicCounts,
      vocabSize: Int,
      eta: Double,
      alpha: Double): TopicCounts = {
    val K = docTopicCounts.length
    val N_j = docTopicCounts.data
    val N_w = termTopicCounts.data
    val N = totalTopicCounts.data
    val eta1 = eta - 1.0
    val alpha1 = alpha - 1.0
    val Weta1 = vocabSize * eta1
    var sum = 0.0
    val gamma_wj = new Array[Double](K)
    var k = 0
    while (k < K) {
      val gamma_wjk = (N_w(k) + eta1) * (N_j(k) + alpha1) / (N(k
) + Weta1)
      gamma_wj(k) = gamma_wjk
      sum += gamma_wjk
      k += 1
    }
    // normalize
    BDV(gamma_wj) /= sum
  }
```

这段代码比较简单，完全按照公式**(3.1.6)**表示的样子来实现。`val gamma_wjk = (N_w(k) + eta1) * (N_j(k) + alpha1) / (N(k) + Weta1)`就是实现的更新逻辑。

4.2.3 M-步：更新phi和theta

```
// M-STEP: 聚合计算新的 N_{kj}, N_{wk} counts.
val docTopicDistributions: VertexRDD[TopicCounts] =
   graph.aggregateMessages[(Boolean, TopicCounts)](sendMsg, mergeMsg).mapValues(_._2)
```

我们由公式**(3.1.7)**可知，更新隐藏变量 `phi` 和 `theta` 就是更新相应的 `N_kj` 和 `N_wk` 。聚合更新使用 `aggregateMessages` 方法来实现。请参考文献【7】来了解该方法的作用。

4.3 在线变分算法的代码实现

4.3.1 初始化状态

在线学习算法首先使用方法 `initialize` 方法初始化参数值

```
override private[clustering] def initialize(
      docs: RDD[(Long, Vector)],
      lda: LDA): OnlineLDAOptimizer = {
    this.k = lda.getK
    this.corpusSize = docs.count()
    this.vocabSize = docs.first()._2.size
    this.alpha = if (lda.getAsymmetricDocConcentration.size == 1
) {
      if (lda.getAsymmetricDocConcentration(0) == -1) Vectors.de
nse(Array.fill(k)(1.0 / k))
      else {
        require(lda.getAsymmetricDocConcentration(0) >= 0,
          s"all entries in alpha must be >=0, got: $alpha")
        Vectors.dense(Array.fill(k)(lda.getAsymmetricDocConcentr
ation(0)))
      }
    } else {
      require(lda.getAsymmetricDocConcentration.size == k,
        s"alpha must have length k, got: $alpha")
      lda.getAsymmetricDocConcentration.foreachActive { case (_,
 x) =>
        require(x >= 0, s"all entries in alpha must be >= 0, got
: $alpha")
      }
      lda.getAsymmetricDocConcentration
    }
    this.eta = if (lda.getTopicConcentration == -1) 1.0 / k else
 lda.getTopicConcentration
    this.randomGenerator = new Random(lda.getSeed)
    this.docs = docs
    // 初始化变分分布 q(beta|lambda)
    this.lambda = getGammaMatrix(k, vocabSize)
    this.iteration = 0
    this
  }
```

根据文献【5】，`alpha` 和 `eta` 的值大于等于0，并且默认为 `1.0/k` 。上文使用 `getGammaMatrix` 方法来初始化变分分布 `q(beta|lambda)` 。

```
private def getGammaMatrix(row: Int, col: Int): BDM[Double] = {
    val randBasis = new RandBasis(new org.apache.commons.math3.r
andom.MersenneTwister(
      randomGenerator.nextLong()))
    //初始化一个gamma分布
    val gammaRandomGenerator = new Gamma(gammaShape, 1.0 / gamma
Shape)(randBasis)
    val temp = gammaRandomGenerator.sample(row * col).toArray
    new BDM[Double](col, row, temp).t
  }
```

`getGammaMatrix` 方法使用 `gamma` 分布初始化一个随机矩阵。

4.3.2 更新参数

```
override private[clustering] def next(): OnlineLDAOptimizer = {
    //返回文档集中采样的子集
    //默认情况下，文档可以被采样多次，且采样比例是0.05
    val batch = docs.sample(withReplacement = sampleWithReplacem
ent, miniBatchFraction,
      randomGenerator.nextLong())
    if (batch.isEmpty()) return this
    submitMiniBatch(batch)
  }
```

以上的 `next` 方法首先对文档进行采样，然后调用 `submitMiniBatch` 对采样的文档子集进行处理。下面我们详细分解 `submitMiniBatch` 方法。

- **1** 计算 `log(beta)` 的期望，并将其作为广播变量广播到集群中

```
val expElogbeta = exp(LDAUtils.dirichletExpectation(lambda)).t
//广播变量
val expElogbetaBc = batch.sparkContext.broadcast(expElogbeta)
//参数alpha是dirichlet参数
private[clustering] def dirichletExpectation(alpha: BDM[Double])
: BDM[Double] = {
    val rowSum = sum(alpha(breeze.linalg.*, ::))
    val digAlpha = digamma(alpha)
    val digRowSum = digamma(rowSum)
    val result = digAlpha(::, breeze.linalg.*) - digRowSum
    result
  }
```

上述代码调用 `exp(LDAUtils.dirichletExpectation(lambda))` 方法实现参数为 `lambda` 的 `log beta` 的期望。实现原理参见公式**(3.2.6)**。

- **2** 计算 `phi` 以及 `gamma` ，即算法**2**中的 `E-步`

```
//对采样文档进行分区处理
val stats: RDD[(BDM[Double], List[BDV[Double]])] = batch.mapPart
itions { docs =>
      //
      val nonEmptyDocs = docs.filter(_._2.numNonzeros > 0)
      val stat = BDM.zeros[Double](k, vocabSize)
      var gammaPart = List[BDV[Double]]()
      nonEmptyDocs.foreach { case (_, termCounts: Vector) =>
        val ids: List[Int] = termCounts match {
          case v: DenseVector => (0 until v.size).toList
          case v: SparseVector => v.indices.toList
        }
        val (gammad, sstats) = OnlineLDAOptimizer.variationalTop
icInference(
          termCounts, expElogbetaBc.value, alpha, gammaShape, k)
        stat(::, ids) := stat(::, ids).toDenseMatrix + sstats
        gammaPart = gammad :: gammaPart
      }
      Iterator((stat, gammaPart))
    }
```

上面的代码调用 `OnlineLDAOptimizer.variationalTopicInference` 实现算法**2**中的 `E-步` ,迭代计算 `phi` 和 `gamma` 。

```
private[clustering] def variationalTopicInference(
      termCounts: Vector,
      expElogbeta: BDM[Double],
      alpha: breeze.linalg.Vector[Double],
      gammaShape: Double,
      k: Int): (BDV[Double], BDM[Double]) = {
    val (ids: List[Int], cts: Array[Double]) = termCounts match
{
      case v: DenseVector => ((0 until v.size).toList, v.values)
      case v: SparseVector => (v.indices.toList, v.values)
    }
    // 初始化变分分布 q(theta|gamma)
    val gammad: BDV[Double] = new Gamma(gammaShape, 1.0 / gammaS
hape).samplesVector(k)   // K
    //根据公式（3.2.6）计算 E(log theta)
    val expElogthetad: BDV[Double] = exp(LDAUtils.dirichletExpec
tation(gammad))  // K
    val expElogbetad = expElogbeta(ids, ::).toDenseMatrix
                 // ids * K
    //根据公式（3.2.5）计算phi，这里加1e-100表示并非严格等于
    val phiNorm: BDV[Double] = expElogbetad * expElogthetad :+ 1
e-100            // ids
    var meanGammaChange = 1D
    val ctsVector = new BDV[Double](cts)
                 // ids
    // 迭代直至收敛
    while (meanGammaChange > 1e-3) {
      val lastgamma = gammad.copy
      //依据公式(3.2.5)计算gamma
      gammad := (expElogthetad :* (expElogbetad.t * (ctsVector :
/ phiNorm))) :+ alpha
      //根据更新的gamma，计算E(log theta)
      expElogthetad := exp(LDAUtils.dirichletExpectation(gammad)
)
      // 更新phi
      phiNorm := expElogbetad * expElogthetad :+ 1e-100
      //计算两次gamma的差值
```

```
        meanGammaChange = sum(abs(gammad - lastgamma)) / k
      }
      val sstatsd = expElogthetad.asDenseMatrix.t * (ctsVector :/
phiNorm).asDenseMatrix
      (gammad, sstatsd)
    }
```

- **3** 更新 `lambda`

```
      val statsSum: BDM[Double] = stats.map(_._1).reduce(_ += _)
      val gammat: BDM[Double] = breeze.linalg.DenseMatrix.vertcat(
        stats.map(_._2).reduce(_ ++ _).map(_.toDenseMatrix): _*)
      val batchResult = statsSum :* expElogbeta.t
      // 更新lambda和alpha
      updateLambda(batchResult, (miniBatchFraction * corpusSize).c
eil.toInt)
```

`updateLambda` 方法实现算法**2**中的 `M-步`,更新 `lambda` 。实现代码如下：

```
private def updateLambda(stat: BDM[Double], batchSize: Int): Unit
 = {
    // 根据公式（3.2.8）计算权重
    val weight = rho()
    // 更新lambda，其中stat * (corpusSize.toDouble / batchSize.toD
ouble)+eta表示rho_cap
    lambda := (1 - weight) * lambda +
      weight * (stat * (corpusSize.toDouble / batchSize.toDouble
) + eta)
  }
// 根据公式（3.2.8）计算rho
private def rho(): Double = {
    math.pow(getTau0 + this.iteration, -getKappa)
  }
```

- **4** 更新 `alpha`

```
private def updateAlpha(gammat: BDM[Double]): Unit = {
    //计算rho
    val weight = rho()
    val N = gammat.rows.toDouble
    val alpha = this.alpha.toBreeze.toDenseVector
    //计算log p_hat
    val logphat: BDM[Double] = sum(LDAUtils.dirichletExpectation
(gammat)(::, breeze.linalg.*)) / N
    //计算梯度为N（-phi(alpha)+log p_hat）
    val gradf = N * (-LDAUtils.dirichletExpectation(alpha) + log
phat.toDenseVector)
    //计算公式（3.3.4）中的c，trigamma表示gamma函数的二阶导数
    val c = N * trigamma(sum(alpha))
    //计算公式（3.3.4）中的q
    val q = -N * trigamma(alpha)
    //根据公式(3.3.7)计算b
    val b = sum(gradf / q) / (1D / c + sum(1D / q))
    val dalpha = -(gradf - b) / q
    if (all((weight * dalpha + alpha) :> 0D)) {
      alpha :+= weight * dalpha
      this.alpha = Vectors.dense(alpha.toArray)
    }
  }
```

5 参考文献

【1】LDA数学八卦

【2】通俗理解LDA主题模型

【3】Latent Dirichlet Allocation

【4】On Smoothing and Inference for Topic Models

【5】Online Learning for Latent Dirichlet Allocation

【6】Spark官方文档

【7】Spark GraphX介绍

【8】Maximum Likelihood Estimation of Dirichlet Distribution Parameters

【9】Variational Bayes

二分 k-means 算法

二分 k-means 算法是分层聚类（Hierarchical clustering）的一种，分层聚类是聚类分析中常用的方法。 分层聚类的策略一般有两种：

- 聚合。这是一种 自底向上 的方法，每一个观察者初始化本身为一类，然后两两结合
- 分裂。这是一种 自顶向下 的方法，所有观察者初始化为一类，然后递归地分裂它们

二分 k-means 算法是分裂法的一种。

1 二分 k-means 的步骤

二分 k-means 算法是 k-means 算法的改进算法，相比 k-means 算法，它有如下优点：

- 二分 k-means 算法可以加速 k-means 算法的执行速度，因为它的相似度计算少了
- 能够克服 k-means 收敛于局部最小的缺点

二分 k-means 算法的一般流程如下所示：

- （1）把所有数据初始化为一个簇，将这个簇分为两个簇。
- （2）选择满足条件的可以分解的簇。选择条件综合考虑簇的元素个数以及聚类代价（也就是误差平方和 SSE ），误差平方和的公式如下所示，其中$w_{(i)}$表示权重值，y^{*}表示该簇所有点的平均值。

$$\mathrm{SSE}=\sum_{1}^{n} w_i(y_i-\mathrm{y}^*)^2$$

- （3）使用 k-means 算法将可分裂的簇分为两簇。
- （4）一直重复（2）（3）步，直到满足迭代结束条件。

以上过程隐含着一个原则是：因为聚类的误差平方和能够衡量聚类性能，该值越小表示数据点越接近于它们的质心，聚类效果就越好。 所以我们就需要对误差平方和最大的簇进行再一次的划分，因为误差平方和越大，表示该簇聚类越不好，越有可能是多个簇被当成一个簇了，所以我们首先需要对这个簇进行划分。

2 二分 `k-means` 的源码分析

`spark` 在文件 `org.apache.spark.mllib.clustering.BisectingKMeans` 中实现了二分 `k-means` 算法。在分步骤分析算法实现之前，我们先来了解 `BisectingKMeans` 类中参数代表的含义。

```
class BisectingKMeans private (
    private var k: Int,
    private var maxIterations: Int,
    private var minDivisibleClusterSize: Double,
    private var seed: Long)
```

上面代码中， `k` 表示叶子簇的期望数，默认情况下为4。如果没有可被切分的叶子簇，实际值会更小。 `maxIterations` 表示切分簇的 `k-means` 算法的最大迭代次数，默认为20。 `minDivisibleClusterSize` 的值如果大于等于1，它表示一个可切分簇的最小点数量；如果值小于1，它表示可切分簇的点数量占总数的最小比例，该值默认为1。

`BisectingKMeans` 的 `run` 方法实现了二分 `k-means` 算法，下面将一步步分析该方法的实现过程。

- （1）初始化数据

```
//计算输入数据的二范式并转化为VectorWithNorm
val norms = input.map(v => Vectors.norm(v, 2.0)).persist(Storage
Level.MEMORY_AND_DISK)
val vectors = input.zip(norms).map { case (x, norm) => new Vecto
rWithNorm(x, norm) }
```

- （2）将所有数据初始化为一个簇，并计算代价

```
var assignments = vectors.map(v => (ROOT_INDEX, v))
var activeClusters = summarize(d, assignments) //格式为Map[index,
ClusterSummary]
val rootSummary = activeClusters(ROOT_INDEX)
```

在上述代码中，第一行给每个向量加上一个索引，用以标明簇在最终生成的树上的深度，`ROOT_INDEX` 的值为1。`summarize` 方法计算误差平方和，我们来看看它的实现。

```
private def summarize(
      d: Int,
      assignments: RDD[(Long, VectorWithNorm)]): Map[Long, Clust
erSummary] = {
    assignments.aggregateByKey(new ClusterSummaryAggregator(d))(
        //分区内循环添加
        seqOp = (agg, v) => agg.add(v),
        //分区间合并
        combOp = (agg1, agg2) => agg1.merge(agg2)
      ).mapValues(_.summary)
      .collect().toMap
}
```

这里的 `d` 表示特征维度，代码对 `assignments` 使用 `aggregateByKey` 操作，根据 `key` 值在分区内循环添加（ `add` ）数据，在分区间合并（ `merge` ）数据集，转换成最终 `ClusterSummaryAggregator` 对象，然后针对每个 `key` ，调用 `summary` 方法，计算。 `ClusterSummaryAggregator` 包含三个很简单的方法，分别是 `add` ， `merge` 以及 `summary` 。

```
private class ClusterSummaryAggregator(val d: Int) extends Seria
lizable {
    private var n: Long = 0L
    private val sum: Vector = Vectors.zeros(d) //向量和
    private var sumSq: Double = 0.0  //向量的范数平方和
    //添加一个VectorWithNorm对象到ClusterSummaryAggregator对象中
    def add(v: VectorWithNorm): this.type = {
      n += 1L
      sumSq += v.norm * v.norm
      BLAS.axpy(1.0, v.vector, sum)
      this
    }
    //合并两个ClusterSummaryAggregator对象
    def merge(other: ClusterSummaryAggregator): this.type = {
      n += other.n
      sumSq += other.sumSq
      //y += a * x
      BLAS.axpy(1.0, other.sum, sum)
      this
    }
    def summary: ClusterSummary = {
      //求平均值
      val mean = sum.copy
      if (n > 0L) {
        //x = a * x
        BLAS.scal(1.0 / n, mean)
      }
      val center = new VectorWithNorm(mean)
      //所有点的范数平方和减去n乘以中心点范数平方，得到误差平方和
      val cost = math.max(sumSq - n * center.norm * center.norm,
0.0)
      new ClusterSummary(n, center, cost)
    }
  }
```

这里计算误差平方和与第一章的公式有所不同，但是效果一致。这里计算聚类代价函数的公式如下所示：

$$SSE=\begin{cases}0, & x<0\\ \sum_{1}^{n}(\|y_i\|_2^2-\|y^*\|_2^2), & x\geq 0\end{cases}$$

获取第一个簇之后，我们需要做的就是迭代分裂可分裂的簇，直到满足我们的要求。迭代停止的条件是 `activeClusters` 为空，或者 `numLeafClustersNeeded` 为0（即没有分裂的叶子簇）,或者迭代深度大于 `LEVEL_LIMIT` 。

```
while (activeClusters.nonEmpty && numLeafClustersNeeded > 0 && l
evel < LEVEL_LIMIT)
```

这里， `LEVEL_LIMIT` 是一个较大的值，计算方法如下。

```
private val LEVEL_LIMIT = math.log10(Long.MaxValue) / math.log10(
2)
```

- （**3**）获取需要分裂的簇

在每一次迭代中，我们首先要做的是获取满足条件的可以分裂的簇。

```
 //选择需要分裂的簇
 var divisibleClusters = activeClusters.filter { case (_, summar
y) =>
    (summary.size >= minSize) && (summary.cost > MLUtils.EPSILON
 * summary.size)
 }
 // If we don't need all divisible clusters, take the larger one
s.
 if (divisibleClusters.size > numLeafClustersNeeded) {
    divisibleClusters = divisibleClusters.toSeq.sortBy { case (_
, summary) =>
        -summary.size
    }.take(numLeafClustersNeeded)
     .toMap
 }
```

这里选择分裂的簇用到了两个条件，即数据点的数量大于规定的最小数量以及代价小于等于 `MLUtils.EPSILON * summary.size` 。并且如果可分解的簇的个数多余我们规定的个数 `numLeafClustersNeeded` 即 `(k-1)` ，那么我们取包含数量最多的 `numLeafClustersNeeded` 个簇用于分裂。

- （4）使用 **`k-means`** 算法将可分裂的簇分解为两簇

我们知道， `k-means` 算法分为两步，第一步是初始化中心点，第二步是迭代更新中心点直至满足最大迭代数或者收敛。下面就分两步来说明。

- 第一步，随机的选择中心点，将可分裂簇分为两簇

```
 //切分簇
var newClusterCenters = divisibleClusters.flatMap { case (index,
 summary) =>
    //随机切分簇为两簇，找到这两个簇的中心点
    val (left, right) = splitCenter(summary.center, random)
    Iterator((leftChildIndex(index), left), (rightChildIndex(ind
ex), right))
}.map(identity)
```

在上面的代码中，用 `splitCenter` 方法将簇随机地分为了两簇，并返回相应的中心点，它的实现如下所示。

```
private def splitCenter(
      center: VectorWithNorm,
      random: Random): (VectorWithNorm, VectorWithNorm) = {
    val d = center.vector.size
    val norm = center.norm
    val level = 1e-4 * norm
    //随机的初始化一个点，并用这个点得到两个初始中心点
    val noise = Vectors.dense(Array.fill(d)(random.nextDouble())
)
    val left = center.vector.copy
    //y += a * x,left=left-level*noise
    BLAS.axpy(-level, noise, left)
    val right = center.vector.copy
    //right=right+level*noise
    BLAS.axpy(level, noise, right)
    //返回中心点
    (new VectorWithNorm(left), new VectorWithNorm(right))
  }
```

- 第二步，迭代更新中心点

```
 var newClusters: Map[Long, ClusterSummary] = null
 var newAssignments: RDD[(Long, VectorWithNorm)] = null
 //迭代获得中心点，默认迭代次数为20
 for (iter <- 0 until maxIterations) {
    //根据更新的中心点，将数据点重新分类
    newAssignments = updateAssignments(assignments, divisibleInd
ices, newClusterCenters)
        .filter { case (index, _) =>
            divisibleIndices.contains(parentIndex(index))
    }
    //计算中心点以及代价值
    newClusters = summarize(d, newAssignments)
    newClusterCenters = newClusters.mapValues(_.center).map(iden
tity)
 }
 val indices = updateAssignments(assignments, divisibleIndices,
newClusterCenters).keys
     .persist(StorageLevel.MEMORY_AND_DISK)
```

这段代码中，`updateAssignments`会根据更新的中心点将数据分配给距离其最短的中心点所在的簇，即重新分配簇。代码如下

```
private def updateAssignments(assignments: RDD[(Long, VectorWith
Norm)],divisibleIndices: Set[Long],
      newClusterCenters: Map[Long, VectorWithNorm]): RDD[(Long,
VectorWithNorm)] = {
    assignments.map { case (index, v) =>
      if (divisibleIndices.contains(index)) {
        //leftChildIndex=2*index , rightChildIndex=2*index+1
        val children = Seq(leftChildIndex(index), rightChildInde
x(index))
        //返回序列中第一个符合条件的最小的元素
        val selected = children.minBy { child =>
          KMeans.fastSquaredDistance(newClusterCenters(child), v
)
        }
        //将v分配给中心点距离其最短的簇
        (selected, v)
      } else {
        (index, v)
      }
    }
  }
```

重新分配簇之后，利用 `summarize` 方法重新计算中心点以及代价值。

- （5）处理变量值为下次迭代作准备

```
//数节点中簇的index以及包含的数据点
 assignments = indices.zip(vectors)
 inactiveClusters ++= activeClusters
 activeClusters = newClusters
 //调整所需簇的数量
 numLeafClustersNeeded -= divisibleClusters.size
```

流式 `k-means` 算法

当数据是以流的方式到达的时候，我们可能想动态的估计（ `estimate` ）聚类的簇，通过新的到达的数据来更新聚类。 `spark.mllib` 支持流式 `k-means` 聚类，并且可以通过参数控制估计衰减（ `decay` ）(或“健忘”(`forgetfulness`))。这个算法使用一般地小批量更新规则来更新簇。

1 流式 `k-means` 算法原理

对每批新到的数据，我们首先将点分配给距离它们最近的簇，然后计算新的数据中心，最后更新每一个簇。使用的公式如下所示：

$$c_{t+1} = \frac{c_t n_t \alpha + x_t m_t}{n_t \alpha + m_t} \qquad (1)$$

$$n_{t+1} = n_t + m_t \qquad (2)$$

在上面的公式中，c_{t}表示前一个簇中心，n_{t}表示分配给这个簇的点的数量， x_{t}表示从当前批数据的簇中心，m_{t}表示当前批数据的点数量。 当评价新的数据时，把衰减因子 `alpha` 当做折扣加权应用到当前的点上，用以衡量当前预测的簇的贡献度量。当 `alpha` 等于1时，所有的批数据赋予相同的权重，当 `alpha` 等于0时，数据中心点完全通过当前数据确定。

衰减因子 `alpha` 也可以通过 `halfLife` 参数联合时间单元（ `time unit` ）来确定，时间单元可以是一批数据也可以是一个数据点。假如数据从 `t` 时刻到来并定义了 `halfLife` 为 `h` ， 在 `t+h` 时刻，应用到 `t` 时刻的数据的折扣（ `discount` ）为0.5。

流式 `k-means` 算法的步骤如下所示：

- （1）分配新的数据点到离其最近的簇；
- （2）根据时间单元（ `time unit` ）计算折扣（ `discount` ）值，并更新簇权重；
- （3）应用更新规则；

- （4）应用更新规则后，有些簇可能消失了，那么切分最大的簇为两个簇。

2 流式 `k-means` 算法源码分析

在分步骤分析源码之前，我们先了解一下 `StreamingKMeans` 参数表达的含义。

```
class StreamingKMeans(
    var k: Int, //簇个数
    var decayFactor: Double,//衰减因子
    var timeUnit: String //时间单元
)
```

在上述定义中，`k` 表示我们要聚类的个数，`decayFactor` 表示衰减因子，用于计算折扣，`timeUnit` 表示时间单元，时间单元既可以是一批数据（`StreamingKMeans.BATCHES`）也可以是单条数据（`StreamingKMeans.POINTS`）。

由于我们处理的是流式数据，所以我们在流式数据来之前要先初始化模型。有两种初始化模型的方法，一种是直接指定初始化中心点及簇权重，一种是随机初始化中心点以及簇权重。

```
//直接初始化中心点及簇权重
def setInitialCenters(centers: Array[Vector], weights: Array[Do
uble]): this.type = {
    model = new StreamingKMeansModel(centers, weights)
    this
}
//随机初始化中心点以及簇权重
def setRandomCenters(dim: Int, weight: Double, seed: Long = Uti
ls.random.nextLong): this.type = {
     val random = new XORShiftRandom(seed)
     val centers = Array.fill(k)(Vectors.dense(Array.fill(dim)(r
andom.nextGaussian())))
     val weights = Array.fill(k)(weight)
     model = new StreamingKMeansModel(centers, weights)
     this
}
```

初始化中心点以及簇权重之后，对于新到的流数据，我们使用更新规则修改中心点和权重，调整聚类情况。更新过程在 `update` 方法中实现，下面我们分步骤分析该方法。

- （1）分配新到的数据到离其最近的簇，并计算更新后的簇的向量和以及点数量

```
 //选择离数据点最近的簇
 val closest = data.map(point => (this.predict(point), (point, 1
L)))
 def predict(point: Vector): Int = {
     //返回和给定点相隔最近的中心
     KMeans.findClosest(clusterCentersWithNorm, new VectorWithNo
rm(point))._1
 }
 // 获得更新的簇的向量和以及点数量
 val mergeContribs: ((Vector, Long), (Vector, Long)) => (Vector,
Long) = (p1, p2) => {
   // y += a * x,向量相加
   BLAS.axpy(1.0, p2._1, p1._1)
   (p1._1, p1._2 + p2._2)
 }
 val pointStats: Array[(Int, (Vector, Long))] = closest
    .aggregateByKey((Vectors.zeros(dim), 0L))(mergeContribs, mer
geContribs)
    .collect()
```

- （2）获取折扣值，并用折扣值作用到权重上

```
 // 折扣
 val discount = timeUnit match {
    case StreamingKMeans.BATCHES => decayFactor
    case StreamingKMeans.POINTS =>
      //所有新增点的数量和
      val numNewPoints = pointStats.view.map { case (_, (_, n))
=>
          n
      }.sum
    // x^y
    math.pow(decayFactor, numNewPoints)
 }
 //将折扣应用到权重上
 //x = a * x
 BLAS.scal(discount, Vectors.dense(clusterWeights))
```

上面的代码更加时间单元的不同获得不同的折扣值。当时间单元为 `StreamingKMeans.BATCHES` 时，折扣就为衰减因子；当时间单元为 `StreamingKMeans.POINTS` 时，折扣由新增数据点的个数 `n` 和衰减因子 `decay` 共同决定。 折扣值为 `n` 个 `decay` 相乘。

- （3）实现更新规则

```
// 实现更新规则
pointStats.foreach { case (label, (sum, count)) =>
   //获取中心点
   val centroid = clusterCenters(label)
   //更新权重
   val updatedWeight = clusterWeights(label) + count
   val lambda = count / math.max(updatedWeight, 1e-16)
   clusterWeights(label) = updatedWeight
   //x = a * x,即（1-lambda）*centroid
   BLAS.scal(1.0 - lambda, centroid)
   // y += a * x，即centroid +=sum*lambda/count
   BLAS.axpy(lambda / count, sum, centroid)
}
```

上面的代码对每一个簇，首先更新簇的权重，权重值为原有的权重加上新增数据点的个数。然后计算 `lambda` ，通过 `lambda` 更新中心点。 `lambda` 为新增数据的个数和更新权重的商。 假设更新之前的中心点为 `c1` ，更新之后的中心点为 `c2` ，那么 `c2=(1-lambda)*c1+sum/count` ，其中 `sum/count` 为所有点的平均值。

- （4）调整权重最小和最大的簇

```
 val weightsWithIndex = clusterWeights.view.zipWithIndex
 //获取权重值最大的簇
 val (maxWeight, largest) = weightsWithIndex.maxBy(_._1)
 //获取权重值最小的簇
 val (minWeight, smallest) = weightsWithIndex.minBy(_._1)
 //判断权重最小的簇是否过小，如果过小，就将这两个簇重新划分为两个新的簇，权
重为两者的均值
 if (minWeight < 1e-8 * maxWeight) {
      logInfo(s"Cluster $smallest is dying. Split the largest cl
uster $largest into two.")
      val weight = (maxWeight + minWeight) / 2.0
      clusterWeights(largest) = weight
      clusterWeights(smallest) = weight
      val largestClusterCenter = clusterCenters(largest)
      val smallestClusterCenter = clusterCenters(smallest)
      var j = 0
      while (j < dim) {
        val x = largestClusterCenter(j)
        val p = 1e-14 * math.max(math.abs(x), 1.0)
        largestClusterCenter.toBreeze(j) = x + p
        smallestClusterCenter.toBreeze(j) = x - p
        j += 1
      }
    }
```

梯度下降算法

梯度下降（ `GD` ）是最小化风险函数、损失函数的一种常用方法，随机梯度下降和批量梯度下降是两种迭代求解思路。

1 批量梯度下降算法

假设 `h(theta)` 是要拟合的函数， `J(theta)` 是损失函数，这里 `theta` 是要迭代求解的值。这两个函数的公式如下,其中 `m` 是训练集的记录条数， `j` 是参数的个数：

$$\mathrm{h}(\theta) = \sum_{j=0}^{n} \theta_j x_j$$

$$\mathrm{J}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (y^i - h_\theta(x_i))^2$$

梯度下降法目的就是求出使损失函数最小时的 `theta` 。批量梯度下降的求解思路如下：

- 对损失函数求 `theta` 的偏导，得到每个 `theta` 对应的的梯度

$$\frac{\rho J(\theta)}{\rho \theta_j} = -\frac{1}{m} \sum_{i=1}^{m} (y^i - h_\theta(x^i)) x_j^i$$

- 按每个参数 `theta` 的梯度负方向，来更新每个 `theta`

$$\theta_j' = \theta_j + \frac{1}{m} \sum_{i=1}^{m} (y^i - h_\theta(x^i)) x_j^i$$

从上面公式可以看到，虽然它得到的是一个全局最优解，但是每迭代一步（即修改 `j` 个 `theta` 参数中的一个），都要用到训练集所有的数据，如果 `m` 很大，迭代速度会非常慢。

2 随机梯度下降算法

随机梯度下降是通过每个样本来迭代更新一次 `theta` ,它大大加快了迭代速度。更新 `theta` 的公式如下所示。

$$\theta_j' = \theta_j + \frac{1}{m}(y^i - h_\theta(x^i))x_j^i$$

批量梯度下降会最小化所有训练样本的损失函数，使得最终求解的是全局的最优解，即求解的参数是使得风险函数最小。随机梯度下降会最小化每条样本的损失函数， 虽然不是每次迭代得到的损失函数都向着全局最优方向， 但是大的整体的方向是向全局最优解的，最终的结果往往是在全局最优解附近。

3 批随机梯度下降算法

在 `MLlib` 中，并不是严格实现批量梯度下降算法和随机梯度下降算法，而是结合了这两种算法。即在每次迭代中，既不是使用所有的样本，也不是使用单个样本，而是抽样一小批样本用于计算。

$$\theta_j' = \theta_j + \frac{1}{m}\sum_{i=1}^{k}(y^i - h_\theta(x^i))x_j^i \text{ }, k \ll m$$

下面分析该算法的实现。首先我们看看 `GradientDescent` 的定义。

```
class GradientDescent private[spark] (private var gradient: Grad
ient, private var updater: Updater)
  extends Optimizer with Logging
```

这里 `Gradient` 类用于计算给定数据点的损失函数的梯度。 `Gradient` 类用于实现参数的更新，即上文中的 `theta` 。梯度下降算法的具体实现在 `runMiniBatchSGD` 中。

```
def runMiniBatchSGD(
      data: RDD[(Double, Vector)],
      gradient: Gradient,
      updater: Updater,
      stepSize: Double,
      numIterations: Int,
      regParam: Double,
      miniBatchFraction: Double,
      initialWeights: Vector,
      convergenceTol: Double): (Vector, Array[Double])
```

这里 `stepSize` 是更新时的步长， `regParam` 表示归一化参数， `miniBatchFraction` 表示采样比例。迭代内部的处理分为两步。

- 1 采样并计算梯度

```
val (gradientSum, lossSum, miniBatchSize) = data.sample(false, miniBatchFraction, 42 + i)
        .treeAggregate((BDV.zeros[Double](n), 0.0, 0L))(
          seqOp = (c, v) => {
            // c: (grad, loss, count), v: (label, features)
            val l = gradient.compute(v._2, v._1, bcWeights.value, Vectors.fromBreeze(c._1))
            (c._1, c._2 + l, c._3 + 1)
          },
          combOp = (c1, c2) => {
            // c: (grad, loss, count)
            (c1._1 += c2._1, c1._2 + c2._2, c1._3 + c2._3)
          })
```

这里 `treeAggregate` 类似于 `aggregate` 方法，不同的是在每个分区，该函数会做两次（默认两次）或两次以上的 `merge` 聚合操作，避免将所有的局部值传回 `driver` 端。

该步按照上文提到的偏导公式求参数的梯度，但是根据提供的 `h` 函数的不同，计算结果会有所不同。 `MLlib` 现在提供的求导方法分别有 `HingeGradient` 、 `LeastSquaresGradient` 、 `LogisticGradient` 以及 `ANNGradient` 。这些类的实现会在具体的算法中介绍。

- 2 更新权重参数

```
val update = updater.compute(
          weights, Vectors.fromBreeze(gradientSum / miniBatchSiz
e.toDouble),
          stepSize, i, regParam)
weights = update._1
regVal = update._2
```

求出梯度之后，我们就可以根据梯度的值更新现有的权重参数。 `MLlib` 现在提供的 `Updater` 主要有 `SquaredL2Updater` 、 `L1Updater` 、 `SimpleUpdater` 等。这些类的实现会在具体的算法中介绍。

参考文献

【1】随机梯度下降和批量梯度下降的公式对比、实现对比

L-BFGS

1 牛顿法

设 `f(x)` 是二次可微实函数，又设$x^{(k)}$是 `f(x)` 一个极小点的估计，我们把 `f(x)` 在$x^{(k)}$处展开成 `Taylor` 级数， 并取二阶近似。

$$f(x) \approx \phi(x) = f(x^{(k)}) + \nabla f(x^{(k)})^T(x - x^k) + \frac{1}{2}(x - x^{(k)})^T \nabla^2 f(x^{(k)})(x - x^{(k)})$$

上式中最后一项的中间部分表示 `f(x)` 在$x^{(k)}$处的 `Hesse` 矩阵。对上式求导并令其等于0，可以的到下式：

$$\nabla f(x^{(k)}) + \nabla^2 f(x^{(k)})(x - x^{(k)}) = 0$$

设 `Hesse` 矩阵可逆，由上式可以得到牛顿法的迭代公式如下**(1.1)**

$$x^{(k+1)} = x^{(k)} - \nabla^2 f(x^{(k)})^{-1} \nabla f(x^{(k)})$$

值得注意 ， 当初始点远离极小点时，牛顿法可能不收敛。原因之一是牛顿方向不一定是下降方向，经迭代，目标函数可能上升。此外，即使目标函数下降，得到的点也不一定是沿牛顿方向最好的点或极小点。 因此，我们在牛顿方向上增加一维搜索，提出阻尼牛顿法。其迭代公式是**(1.2)**：

$$x^{(k+1)} = x^{(k)} + \lambda_k d^{(k)},$$
$$d^{(k)} = -\nabla^2 f(x^{(k)})^{-1} \nabla f(x^{(k)})$$

其中， `lambda` 是由一维搜索（参考文献【1】了解一维搜索）得到的步长，即满足

$$f(x^{(k)} + \lambda_k d^{(k)}) = \min f(x^{(k)} + \lambda d^{(k)})$$

2 拟牛顿法

2.1 拟牛顿条件

前面介绍了牛顿法，它的突出优点是收敛很快，但是运用牛顿法需要计算二阶偏导数，而且目标函数的 `Hesse` 矩阵可能非正定。为了克服牛顿法的缺点，人们提出了拟牛顿法，它的基本思想是用不包含二阶导数的矩阵近似牛顿法中的 `Hesse` 矩阵的逆矩阵。 由于构造近似矩阵的方法不同，因而出现不同的拟牛顿法。

下面分析怎样构造近似矩阵并用它取代牛顿法中的 `Hesse` 矩阵的逆。上文 **(1.2)** 已经给出了牛顿法的迭代公式，为了构造 `Hesse` 矩阵逆矩阵的近似矩阵 $H_{(k)}$ ，需要先分析该逆矩阵与一阶导数的关系。

设在第 `k` 次迭代之后，得到$x^{(k+1)}$ ，我们将目标函数 `f(x)` 在点$x^{(k+1)}$展开成 `Taylor` 级数， 并取二阶近似，得到

$$f(x) \approx f(x^{(k+1)}) + \nabla f(x^{(k+1)})^T(x - x^k) + \frac{1}{2}(x - x^{(k+1)})^T \nabla^2 f(x^{(k+1)})(x - x^{(k+1)})$$

由此可知，在$x^{(k+1)}$附近有，

$$\nabla f(x) \approx \nabla f(x^{(k+1)}) + \nabla^2 f(x^{(k+1)})(x - x^{(k+1)})$$
$$\nabla f(x^{(k)}) \approx \nabla f(x^{(k+1)}) + \nabla^2 f(x^{(k+1)})(x^{(k)} - x^{(k+1)})$$

记

$$p^{(k)} = x^{(k+1)} - x^{(k)}$$
$$q^{(k)} = \nabla f(x^{(k+1)}) - \nabla f(x^{(k)})$$

则有

$$q^{(k)} \approx \nabla^2 f(x^{(k+1)}) p^{(k)}$$

又设 `Hesse` 矩阵可逆，那么上式可以写为如下形式。

$$p^{(k)} \approx \nabla^2 f(x^{(k+1)})^{-1} q^{(k)}$$

这样，计算出 `p` 和 `q` 之后，就可以通过上面的式子估计 `Hesse` 矩阵的逆矩阵。因此，为了用不包含二阶导数的矩阵$H*{(k+1)}$*取代牛顿法中 `Hesse` 矩阵的逆矩阵，有理由令*$H*{(k+1)}$满足公式**(2.1)**：

$$p^{(k)} = H_{k+1} q^{(k)}$$

公式**(2.1)**称为拟牛顿条件。

2.2 秩1校正

当 `Hesse` 矩阵的逆矩阵是对称正定矩阵时，满足拟牛顿条件的矩阵$H*{(k)}$*也应该是对称正定矩阵。构造这样近似矩阵的一般策略是，*$H*{(1)}$取为任意一个 `n` 阶对称正定矩阵，通常选择 `n` 阶单位矩阵 `I` ，然后通过修正$H*{(k)}$*给定*$H*{(k+1)}$。 令，

$$H_{k+1} = H_k + \Delta H_k$$

秩1校正公式写为如下公式**(2.2)**形式。

$$H_{k+1} = H_k + \frac{(p^{(k)} - H_k q^{(k)})(p^{(k)} - H_k q^{(k)})^T}{q^{(k)T}(p^{(k)} - H_k q^{(k)})}$$

2.3 DFP算法

著名的 `DFP` 方法是 `Davidon` 首先提出，后来又被 `Feltcher` 和 `Powell` 改进的算法，又称为变尺度法。在这种方法中，定义校正矩阵为公式**(2.3)**

$$\Delta H_k = \frac{p^{(k)} p^{(k)T}}{p^{(k)T} q^{(k)}} - \frac{H_k q^{(k)} q^{(k)T} H_k}{q^{(k)T} H_k q^{(k)}}$$

那么得到的满足拟牛顿条件的 `DFP` 公式如下**(2.4)**

$$H_{k+1} = H_k + \frac{p^{(k)}p^{(k)T}}{p^{(k)T}q^{(k)}} - \frac{H_k q^{(k)} q^{(k)T} H_k}{q^{(k)T} H_k q^{(k)}}$$

查看文献【1】，了解 `DFP` 算法的计算步骤。

2.4 BFGS算法

前面利用拟牛顿条件**(2.1)**推导出了 `DFP` 公式**(2.4)**。下面我们用不含二阶导数的矩阵$B_{(k+1)}$近似 `Hesse` 矩阵，从而给出另一种形式的拟牛顿条件**(2.5)**:

$$q^{(k)} = B_{k+1} p^{(k)}$$

将公式**(2.1)**的 `H` 换为 `B` ， `p` 和 `q` 互换正好可以得到公式**(2.5)**。所以我们可以得到 `B` 的修正公式**(2.6)**:

$$B_{k+1} = B_k + \frac{q^{(k)}q^{(k)T}}{q^{(k)T}p^{(k)}} - \frac{B_k p^{(k)} p^{(k)T} B_k}{p^{(k)T} B_k p^{(k)}}$$

这个公式称关于矩阵 `B` 的 `BFGS` 修正公式，也称为 `DFP` 公式的对偶公式。设$B_{(k+1)}$可逆，由公式**(2.1)**以及**(2.5)**可以推出：

$$H_{k+1} = B_{k+1}^{-1}$$

这样可以得到关于 `H` 的 `BFGS` 公式为下面的公式**(2.7)**:

$$H_{k+1}^{BFGS} = H_k + \left(1 + \frac{q^{(k)T} H_k q^{(k)}}{p^{(k)T} q^{(k)}}\right) \frac{p^{(k)} p^{(k)T}}{p^{(k)T} q^{(k)}} - \frac{p^{(k)} q^{(k)T} H_k + H_k q^{(k)} p^{(k)T}}{p^{(k)T} q^{(k)}}$$

这个重要公式是由 `Broyden` , `Fletcher` , `Goldfard` 和 `Shanno` 于1970年提出的，所以简称为 `BFGS` 。数值计算经验表明，它比 `DFP` 公式还好，因此目前得到广泛应用。

2.5 L-BFGS（限制内存BFGS）算法

在 `BFGS` 算法中，仍然有缺陷，比如当优化问题规模很大时，矩阵的存储和计算将变得不可行。为了解决这个问题，就有了 `L-BFGS` 算法。 `L-BFGS` 即 `Limited-memory BFGS` 。 `L-BFGS` 的基本思想是只保存最近的 `m` 次迭代信息，从而大大减少数据的存储空间。对照 `BFGS` ，重新整理一下公式：

$$
\begin{aligned}
s_k &= x_k - x_{k-1} \\
\mathrm{t}_k &= \nabla f(x_k) - \nabla f(x_{k-1}) \\
\rho_k &= \frac{1}{t_k^T s_k} \\
\mathrm{V}_k &= I - \rho_k \mathrm{t}_k s_k^T
\end{aligned}
$$

之前的 `BFGS` 算法有如下公式**(2.8)**

$$
B_{i+1} = V_i^T B_i V_i + \rho_i s_i s_i^T \qquad \text{其中} \rho_i = \frac{1}{t_i^T s_i}, \; V_i = I - \rho_i t_i s_i^T
$$

那么同样有

$$
B_i = V_{i-1}^T B_{i-1} V_{i-1} + \rho_{i-1} s_{i-1} s_{i-1}^T
$$

将该式子带入到公式**(2.8)**中，可以推导出如下公式

$$
\begin{aligned}
B_{i+1} &= V_i^T B_i V_i + \rho_i s_i s_i^T \\
&= V_i^T \left(V_{i-1}^T B_{i-1} V_{i-1} + \rho_{i-1} s_{i-1} s_{i-1}^T \right) V_i + \rho_i s_i s_i^T \\
&= V_i^T V_{i-1}^T B_{i-1} V_{i-1} V_i + V_i^T \rho_{i-1} s_{i-1} s_{i-1}^T V_i + \rho_i s_i s_i^T
\end{aligned}
$$

假设当前迭代为 `k` ，只保存最近的 `m` 次迭代信息，按照上面的方式迭代 `m` 次，可以得到如下的公式**(2.9)**

$$B_i = \left(V_{i-1}^T \dots V_{i-m}^T\right) B_i^0 \left(V_{i-m}^T \dots V_{i-1}^T\right)$$
$$+ \rho_{i-m} \left(V_{i-1}^T \dots V_{i-m+1}^T\right) s_{i-m} s_{i-m}^T \left(V_{i-m+1} \dots V_{i-1}\right)$$
$$+ \rho_{i-m+1} \left(V_{i-1}^T \dots V_{i-m+2}^T\right) s_{i-m+1} s_{i-m+1}^T \left(V_{i-m+2} \dots V_{i-1}\right)$$
$$+ \dots$$
$$+ \rho_{i-1} s_{i-1} s_{i-1}^T$$

上面迭代的最终目的就是找到 `k` 次迭代的可行方向，即

$$\mathrm{r}_k = -B_k \nabla f(x_k)$$

为了求可行方向 `r` ，可以使用 `two-loop recursion` 算法来求。该算法的计算过程如下，算法中出现的 `y` 即上文中提到的 `t` ：

1) IF ITER $\leqslant M$ SET INCR = 0; BOUND = ITER ELSE SET INCR = ITER − M BOUND = M

2) $q_{\text{BOUND}} = g_{\text{ITER}}$

3) FOR i = (BOUND−1), . . . , 0

$$\left[\begin{array}{lr} j = i + \text{incr} & \\ \alpha_i = \rho_j s_j^T q_{i+1} \quad (\text{STORE } \alpha_i) & \text{(a)} \\ q_i = q_{i+1} - \alpha_i y_j & \text{(b)} \end{array}\right.$$

$r_0 = H_0 \cdot q_0$ (c)

FOR i = 0, 1, . . . , (BOUND−1)

$$\left[\begin{array}{lr} j = i + \text{incr} & \\ \beta_j = \rho_j y_j^T r_i & \text{(d)} \\ r_{i+1} = r_i + s_j(\alpha_i - \beta_i) & \text{(e)} \end{array}\right.$$

算法 `L-BFGS` 的步骤如下所示。

Step 1：选初始点x_0，运行误差$\varepsilon > 0$，存储最近 m 次的迭代数据；

Step 2：$k = 0, H_0 = I, r = \nabla f(x_0)$；

Step 3：如果$||\nabla f(x_{k+1})|| \le \varepsilon$，则返回最优解 x，否则转入 Step 4；

Step 4：计算本次迭代的可行方向$p_k = -r_k$；

Step 5：计算步长$\alpha_k > 0$，对下面的式子进行一维搜索

$$f(x_k + \alpha_k p_k) = \min f(x_k + \alpha p_k)$$

Step 6：更新权重 x

$$x_{k+1} = x_k + \alpha_k p_k$$

Step 7：如果 k 大于 m，保留最近 m 次的向量对，删除(s_{k-m}, t_{k-m})

Step 8：计算并保持

$$s_k = x_{k+1} - x_k$$

$$t_k = \nabla f(x_{k+1}) - \nabla f(x_k)$$

Step 9：用 two-loop recursion 算法求$r_k = B_k \nabla f(x_k)$

Step 10：k=k+1，并转 Step 3

2.6 OWL-QN算法

2.6.1 L1 正则化

在机器学习算法中，使用损失函数作为最小化误差，而最小化误差是为了让我们的模型拟合我们的训练数据，此时， 若参数过分拟合我们的训练数据就会有过拟合的问题。正则化参数的目的就是为了防止我们的模型过分拟合训练数据。此时，我们会在损失项之后加上正则化项以约束模型中的参数：

$$J(x) = l(x) + r(x)$$

公式右边的第一项是损失函数，用来衡量当训练出现偏差时的损失，可以是任意可微凸函数（如果是非凸函数该算法只保证找到局部最优解）。 第二项是正则化项。用来对模型空间进行限制，从而得到一个更“简单”的模型。

根据对模型参数所服从的概率分布的假设的不同，常用的正则化一般有 `L2` 正则化（模型参数服从 `Gaussian` 分布）、 `L1` 正则化（模型参数服从 `Laplace` 分布）以及它们的组合形式。

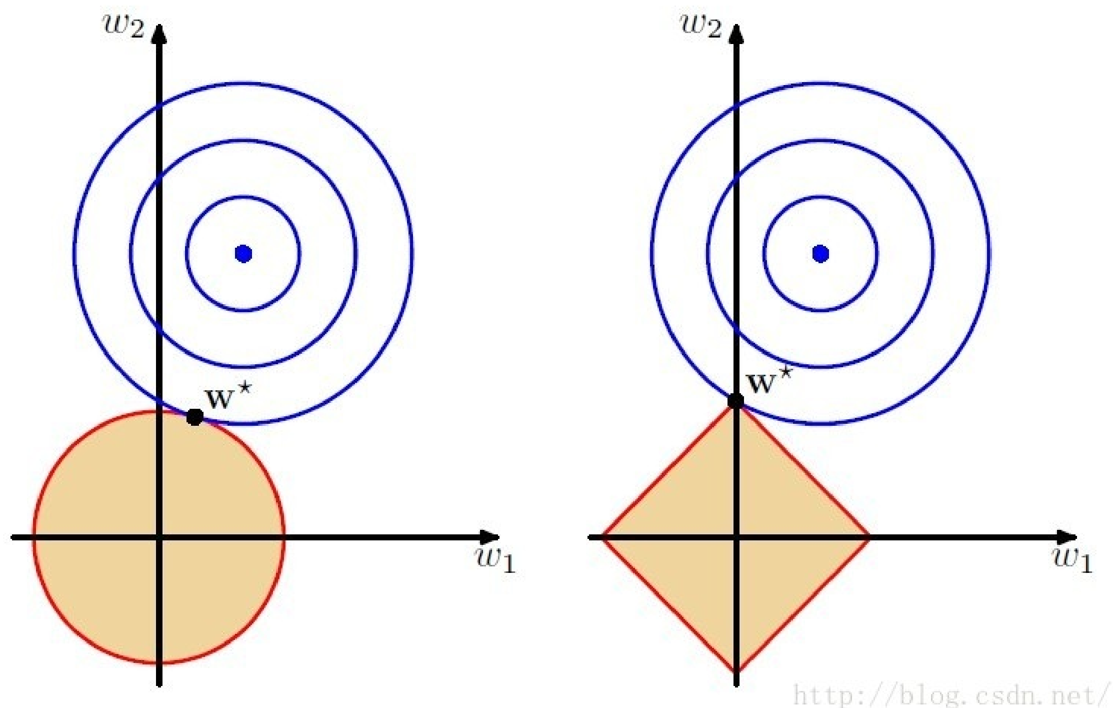
`L1` 正则化的形式如下

$$J(x) = l(x) + C ||x||_{1}$$

`L2` 正则化的形式如下

$$J(x) = l(x) + C ||x||_{2}$$

`L1` 正则化和 `L2` 正则化之间的一个最大区别在于前者可以产生稀疏解，这使它同时具有了特征选择的能力，此外，稀疏的特征权重更具有解释意义。如下图：



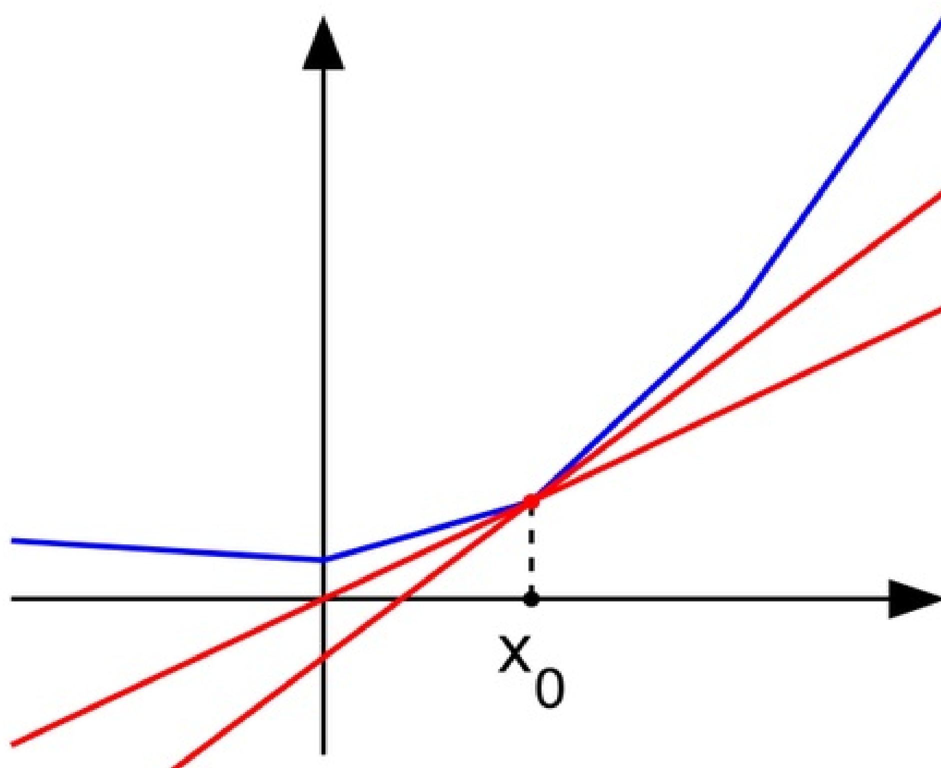
图左侧是 `L2` 正则，右侧为 `L1` 正则。当模型中只有两个参数，即w_1和w_2时， `L2` 正则的约束空间是一个圆，而 `L1` 正则的约束空间为一个正方形，这样，基于 `L1` 正则的约束会产生稀疏解，即图中某一维(w_2)为0。 而 `L2` 正则只是将参数约束在接近0的很小的区间里，而不会正好为0(不排除有0的情况)。对于 `L1` 正则产生的稀疏解有很多的好处，如可以起到特征选择的作用，因为有些维的系数为0，说明这些维对于模型的作用很小。

这里有一个问题是， `L1` 正则化项不可微，所以无法像求 `L-BFGS` 那样去求。微软提出了 `OWL-QN` (`Orthant-Wise Limited-Memory Quasi-Newton`)算法，该算法是基于 `L-BFGS` 算法的可用于求解 `L1` 正则的算法。 简单来讲， `OWL-QN` 算法是指假定变量的象限确定的条件下使用 `L-BFGS` 算法来更新，同时，使得更新前后变量在同一个象限中(使用映射来满足条件)。

2.6.2 OWL-QN算法的具体过程

- **1** 次微分

设$f:I\rightarrow R$是一个实变量凸函数，定义在实数轴上的开区间内。这种函数不一定是处处可导的，例如绝对值函数$f(x)=|x|$。但是，从下面的图中可以看出（也可以严格地证明），对于定义域中的任何x_0，我们总可以作出一条直线，它通过点(x_0, $f(x_0)$)，并且要么接触f的图像，要么在它的下方。 这条直线的斜率称为函数的次导数。推广到多元函数就叫做次梯度。



凸函数$f:I\rightarrow R$在点x_0的次导数，是实数 `c` 使得：

$$f(x)-f(x_0)\geq c(x-x_0)$$

对于所有 `I` 内的 `x` 。我们可以证明，在点x_0的次导数的集合是一个非空闭区间$[a, b]$，其中 `a` 和 `b` 是单侧极限。

$$a=\lim_{x\to x_0^-}\frac{f(x)-f(x_0)}{x-x_0}$$

$$b=\lim_{x\to x_0^+}\frac{f(x)-f(x_0)}{x-x_0}$$

它们一定存在，且满足$a \leqslant b$。所有次导数的集合$[a, b]$称为函数 `f` 在x_0的次微分。

- **2** 伪梯度

利用次梯度的概念推广了梯度，定义了一个符合上述原则的伪梯度，求一维搜索的可行方向时用伪梯度来代替 `L-BFGS` 中的梯度。

$$\diamond_i f(x) = \begin{cases} \partial_i^- f(x) & \text{if } \partial_i^- f(x) > 0 \\ \partial_i^+ f(x) & \text{if } \partial_i^+ f(x) < 0 \\ 0 & \text{otherwise,} \end{cases}$$

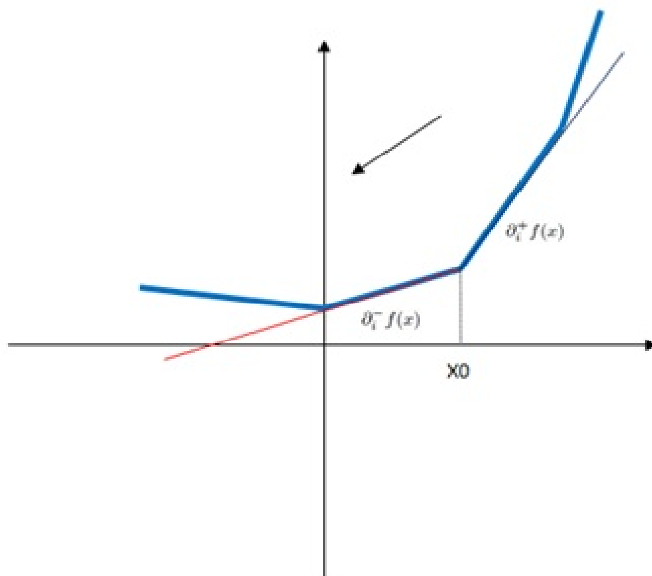
其中

$$\partial_i^{\pm} f(x) = \frac{\partial}{\partial x_i}\ell(x) + \begin{cases} C\sigma(x_i) & \text{if } x_i \neq 0 \\ \pm C & \text{if } x_i = 0. \end{cases}$$

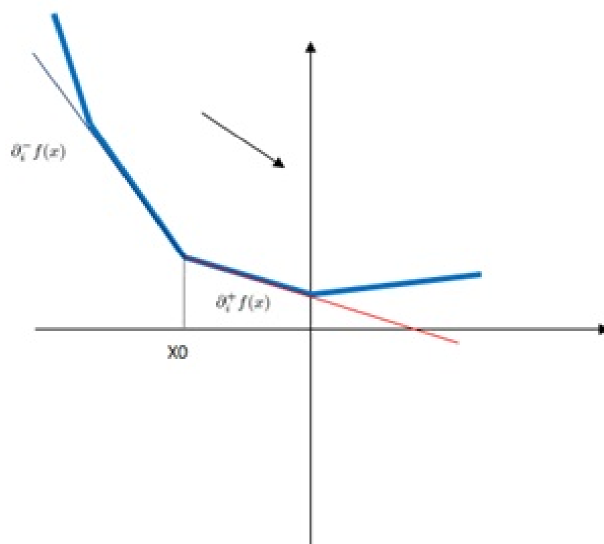
$$\partial_i^- f(x) \leq \partial_i^+ f(x)$$

我们要如何理解这个伪梯度呢？对于不是处处可导的凸函数，可以分为下图所示的三种情况。

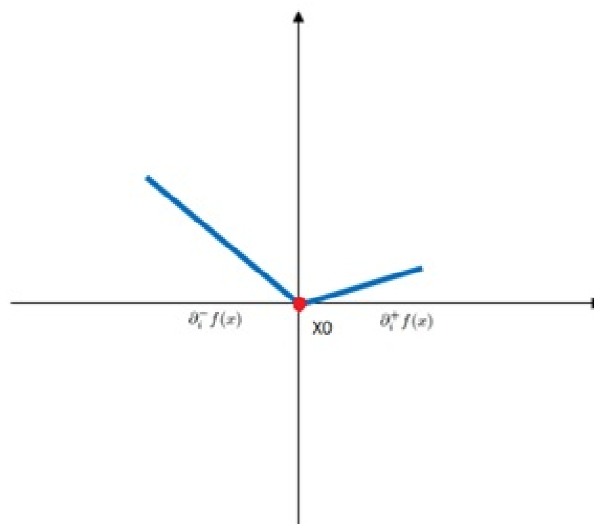
左侧极限小于0：



右侧极限大于0：



其它情况：



结合上面的三幅图表示的三种情况以及伪梯度函数公式，我们可以知道，伪梯度函数保证了在x_0处取得的方向导数是最小的。

- **3** 映射

有了函数的下降的方向，接下来必须对变量的所属象限进行限制，目的是使得更新前后变量在同一个象限中，定义函数：$\pi: \mathbb{R}^{n} \rightarrow \mathbb{R}^{n}$

$$\pi_i(x; y) = \begin{cases} x_i & \text{if } \sigma(x_i) = \sigma(y_i), \\ 0 & \text{otherwise,} \end{cases}$$

上述函数π直观的解释是若x和y在同一象限则取x，若两者不在同一象限中，则取0。

- **4** 线搜索

上述的映射是防止更新后的变量的坐标超出象限，而对坐标进行的一个约束，具体的约束的形式如下：

$$x^{k+1} = \pi\left(x^k + \alpha p^k;\ \xi\right)$$

其中$x^{k} + \alpha p _{k}$是更新公式，ζ表示x^k所在的象限，p^k表示伪梯度下降的方向，它们具体的形式如下：

$$\xi_i = \begin{cases} \sigma\left(x_i^k\right) & \text{if } x_i^k \neq 0 \\ \sigma\left(-\Diamond_i f\left(x^k\right)\right) & \text{if } x_i^k = 0 \end{cases}$$

$$p^k = \pi\left(d^k;\ v^k\right)$$

上面的公式中，v^k为负伪梯度方向，$d^k = H_{k}v^{k}$。

选择α的方式有很多种，在 `OWL-QN` 中，使用了 `backtracking line search` 的一种变种。选择常数$\beta, \gamma \subset (0,1)$，对于$n=0,1,2,...$，使得 $\alpha = \beta^{n}$满足：

$$f\left(\pi\left(x^k + \alpha p^k;\ \xi\right)\right) \leq f\left(x^k\right) - \gamma v^T\left[\pi\left(x^k + \alpha p^k;\ \xi\right) - x^k\right]$$

- **5** 算法流程

Algorithm 1 OWL-QN

choose initial point x^0
$S \Leftarrow \{\}, Y \Leftarrow \{\}$
for $k = 0$ **to** MaxIters **do**
  Compute $v^k = -\diamond f(x^k)$ (1)
  Compute $d^k \Leftarrow \boldsymbol{H}_k v^k$ using S and Y
  $p^k \Leftarrow \pi(d^k; v^k)$ (2)
  Find x^{k+1} with constrained line search (3)
  if termination condition satisfied **then**
    Stop and return x^{k+1}
  end if
  Update S with $s^k = x^{k+1} - x^k$
  Update Y with $y^k = \nabla\ell(x^{k+1}) - \nabla\ell(x^k)$ (4)
end for

与 `L-BFGS` 相比，第一步用伪梯度代替梯度，第二、三步要求一维搜索不跨象限，也就是迭代前的点与迭代后的点处于同一象限，第四步要求估计 `Hessian` 矩阵时依然使用损失函数的梯度。

3 源码解析

3.1 BreezeLBFGS

`spark Ml` 调用 `breeze` 中实现的 `BreezeLBFGS` 来解最优化问题。

```
val optimizer = new BreezeLBFGS[BDV[Double]]($(maxIter), 10, $(t
ol))
val states =
      optimizer.iterations(new CachedDiffFunction(costFun), init
ialWeights.toBreeze.toDenseVector)
```

下面重点分析 `lbfgs.iterations` 的实现。

```
def iterations(f: DF, init: T): Iterator[State] = {
    val adjustedFun = adjustFunction(f)
    infiniteIterations(f, initialState(adjustedFun, init)).takeU
pToWhere(_.converged)
}
```

```
//调用infiniteIterations，其中State是一个样本类
def infiniteIterations(f: DF, state: State): Iterator[State] = {
    var failedOnce = false
    val adjustedFun = adjustFunction(f)
    //无限迭代
    Iterator.iterate(state) { state => try {
        //1 选择梯度下降方向
        val dir = chooseDescentDirection(state, adjustedFun)
        //2 计算步长
        val stepSize = determineStepSize(state, adjustedFun, dir
)
        //3 更新权重
        val x = takeStep(state,dir,stepSize)
        //4 利用CostFun.calculate计算损失值和梯度
        val (value,grad) = calculateObjective(adjustedFun, x, st
ate.history)
        val (adjValue,adjGrad) = adjust(x,grad,value)
        val oneOffImprovement = (state.adjustedValue - adjValue)
/(state.adjustedValue.abs max adjValue.abs max 1E-6 * state.init
ialAdjVal.abs)
        //5 计算s和t
        val history = updateHistory(x,grad,value, adjustedFun, s
tate)
        //6 只保存m个需要的s和t
        val newAverage = updateFValWindow(state, adjValue)
        failedOnce = false
        var s = State(x,value,grad,adjValue,adjGrad,state.iter +
1, state.initialAdjVal, history, newAverage, 0)
        val improvementFailure = (state.fVals.length >= minImpro
vementWindow && state.fVals.nonEmpty && state.fVals.last > state
.fVals.head * (1-improvementTol))
        if(improvementFailure)
          s = s.copy(fVals = IndexedSeq.empty, numImprovementFai
lures = state.numImprovementFailures + 1)
        s
      } catch {
        case x: FirstOrderException if !failedOnce =>
          failedOnce = true
          logger.error("Failure! Resetting history: " + x)
          state.copy(history = initialHistory(adjustedFun, state
```

```
.x))
        case x: FirstOrderException =>
          logger.error("Failure again! Giving up and returning.
Maybe the objective is just poorly behaved?")
          state.copy(searchFailed = true)
      }
    }
  }
```

看上面的代码注释，它的流程可以分五步来分析。

3.1.1 选择梯度下降方向

```
protected def chooseDescentDirection(state: State, fn: DiffFunct
ion[T]):T = {
    state.history * state.grad
}
```

这里的 `*` 是重写的方法，它的实现如下：

```
def *(grad: T) = {
      val diag = if(historyLength > 0) {
        val prevStep = memStep.head
        val prevGradStep = memGradDelta.head
        val sy = prevStep dot prevGradStep
        val yy = prevGradStep dot prevGradStep
        if(sy < 0 || sy.isNaN) throw new NaNHistory
        sy/yy
      } else {
        1.0
      }
      val dir = space.copy(grad)
      val as = new Array[Double](m)
      val rho = new Array[Double](m)
      //第一次递归
      for(i <- 0 until historyLength) {
        rho(i) = (memStep(i) dot memGradDelta(i))
        as(i) = (memStep(i) dot dir)/rho(i)
        if(as(i).isNaN) {
          throw new NaNHistory
        }
        axpy(-as(i), memGradDelta(i), dir)
      }
      dir *= diag
      //第二次递归
      for(i <- (historyLength - 1) to 0 by (-1)) {
        val beta = (memGradDelta(i) dot dir)/rho(i)
        axpy(as(i) - beta, memStep(i), dir)
      }
      dir *= -1.0
      dir
     }
  }
```

非常明显，该方法就是实现了上文提到的 `two-loop recursion` 算法。

3.1.2 计算步长

```
protected def determineStepSize(state: State, f: DiffFunction[T]
, dir: T) = {
    val x = state.x
    val grad = state.grad
    val ff = LineSearch.functionFromSearchDirection(f, x, dir)
    val search = new StrongWolfeLineSearch(maxZoomIter = 10, max
LineSearchIter = 10) // TODO: Need good default values here.
    val alpha = search.minimize(ff, if(state.iter == 0.0) 1.0/no
rm(dir) else 1.0)
    if(alpha * norm(grad) < 1E-10)
      throw new StepSizeUnderflow
    alpha
  }
```

这一步对应 `L-BFGS` 的步骤的 `Step 5` ，通过一维搜索计算步长。

3.1.3 更新权重

```
protected def takeStep(state: State, dir: T, stepSize: Double) =
 state.x + dir * stepSize
```

这一步对应 `L-BFGS` 的步骤的 `Step 5` ，更新权重。

3.1.4 计算损失值和梯度

```
 protected def calculateObjective(f: DF, x: T, history: History)
: (Double, T) = {
     f.calculate(x)
  }
```

这一步对应 `L-BFGS` 的步骤的 `Step 7` ，使用传入的 `CostFun.calculate` 方法计算梯度和损失值。并计算出 `s` 和 `t` 。

3.1.5 计算s和t，并更新history

```
//计算s和t
protected def updateHistory(newX: T, newGrad: T, newVal: Double,
  f: DiffFunction[T], oldState: State): History = {
    oldState.history.updated(newX - oldState.x, newGrad :- oldSt
ate.grad)
}
//添加新的s和t，并删除过期的s和t
protected def updateFValWindow(oldState: State, newAdjVal: Double
):IndexedSeq[Double] = {
    val interm = oldState.fVals :+ newAdjVal
    if(interm.length > minImprovementWindow) interm.drop(1)
    else interm
  }
```

3.2 BreezeOWLQN

`BreezeOWLQN` 的实现与 `BreezeLBFGS` 的实现主要有下面一些不同点。

3.2.1 选择梯度下降方向

```
override protected def chooseDescentDirection(state: State, fn:
DiffFunction[T]) = {
    val descentDir = super.chooseDescentDirection(state.copy(gra
d = state.adjustedGradient), fn)

    // The original paper requires that the descent direction be
 corrected to be
    // in the same directional (within the same hypercube) as th
e adjusted gradient for proof.
    // Although this doesn't seem to affect the outcome that muc
h in most of cases, there are some cases
    // where the algorithm won't converge (confirmed with the au
thor, Galen Andrew).
    val correctedDir = space.zipMapValues.map(descentDir, state.
adjustedGradient, { case (d, g) => if (d * g < 0) d else 0.0 })

    correctedDir
  }
```

此处调用了 `BreezeLBFGS` 的 `chooseDescentDirection` 方法选择梯度下降的方向，然后调整该下降方向为正确的方向（方向必须一致）。

3.2.2 计算步长α

```
override protected def determineStepSize(state: State, f: DiffFunction[T], dir: T) = {
    val iter = state.iter

    val normGradInDir = {
      val possibleNorm = dir dot state.grad
      possibleNorm
    }
    val ff = new DiffFunction[Double] {
       def calculate(alpha: Double) = {
         val newX = takeStep(state, dir, alpha)
         val (v, newG) =  f.calculate(newX)  // 计算梯度
         val (adjv, adjgrad) = adjust(newX, newG, v) // 调整梯度
         adjv -> (adjgrad dot dir)
       }
    }
    val search = new BacktrackingLineSearch(state.value, shrinkStep= if(iter < 1) 0.1 else 0.5)
    val alpha = search.minimize(ff, if(iter < 1) .5/norm(state.grad) else 1.0)

    alpha
  }
```

`takeStep` 方法用于更新参数。

```
  // projects x to be on the same orthant as y
  // this basically requires that x'_i = x_i if sign(x_i) == sig
n(y_i), and 0 otherwise.

  override protected def takeStep(state: State, dir: T, stepSize
: Double) = {
    val stepped = state.x + dir * stepSize
    val orthant = computeOrthant(state.x, state.adjustedGradient
)
    space.zipMapValues.map(stepped, orthant, { case (v, ov) =>
      v * I(math.signum(v) == math.signum(ov))
    })
  }
```

`calculate`方法用于计算梯度，`adjust`方法用于调整梯度。

```
// Adds in the regularization stuff to the gradient
  override protected def adjust(newX: T, newGrad: T, newVal: Dou
ble): (Double, T) = {
    var adjValue = newVal
    val res = space.zipMapKeyValues.mapActive(newX, newGrad, {ca
se (i, xv, v) =>
      val l1regValue = l1reg(i)
      require(l1regValue >= 0.0)

      if(l1regValue == 0.0) {
        v
      } else {
        adjValue += Math.abs(l1regValue * xv)
        xv match {
          case 0.0 => {
            val delta_+ = v + l1regValue   //计算左导数
            val delta_- = v - l1regValue   //计算右导数
            if (delta_- > 0) delta_- else if (delta_+ < 0) delta
_+ else 0.0
          }
          case _ => v + math.signum(xv) * l1regValue
        }
      }
    })
    adjValue -> res
  }
```

参考文献

【1】陈宝林，最优化理论和算法

【2】Updating Quasi-Newton Matrices with Limited Storage

【3】On the Limited Memory BFGS Method for Large Scale Optimization

【4】L-BFGS算法

【5】BFGS算法

【6】逻辑回归模型及LBFGS的Sherman Morrison(SM) 公式推导

【7】Scalable Training of L1-Regularized Log-Linear Models

非负最小二乘

`spark` 中的非负正则化最小二乘法并不是 `wiki` 中介绍的NNLS的实现，而是做了相应的优化。它使用改进投影梯度法结合共轭梯度法来求解非负最小二乘。在介绍 `spark` 的源码之前，我们要先了解什么是最小二乘法以及共轭梯度法。

1 最小二乘法

1.1 最小二乘问题

在某些最优化问题中，目标函数由若干个函数的平方和构成，它的一般形式如下所示：

$$\mathrm{F(x)} = \sum_{i=1}^{m} f_i^2(x), \qquad (1.1)$$

其中 `x=（x1,x2,…,xn）` ，一般假设 `m>=n` 。把极小化这类函数的问题称为最小二乘问题。

$$\min \quad \mathrm{F(x)} = \sum_{i=1}^{m} f_i^2(x), \qquad (1.2)$$

当$f*{i}(x)$为 `x` 的线性函数时，称（*1.2*）为线性最小二乘问题，当*$f{i}*(x)$为 `x` 的非线性函数时，称（1.2）为非线性最小二乘问题。

1.2 线性最小二乘问题

在公式（1.1）中，假设

$$f_i(x) = p_i^T x - b_i \quad , \qquad i = 1, \ldots, i \qquad (1.3)$$

其中， `p` 是n维列向量， `bi` 是实数，这样我们可以用矩阵的形式表示（1.1）式。令

$$A=\begin{bmatrix} p_1^T \\ \vdots \\ p_m^T \end{bmatrix},\quad b=\begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$$

A是 $m * n$ 矩阵， b 是 m 维列向量。则

$$\begin{aligned}F(x) &= \sum_{i=1}^{m} f_i^2(x) = \left(f_1(x), f_2(x), \ldots, f_m(x)\right)\begin{bmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{bmatrix} = (Ax-b)^T(Ax-b) \\ &= x^TA^TAx - 2b^TAx + b^Tb \qquad (1.4)\end{aligned}$$

因为 $F(x)$ 是凸的，所以对（1.4）求导可以得到全局极小值，令其导数为0，我们可以得到这个极小值。

$$\begin{aligned}&\nabla F(x) = 2A^TAx - 2A^Tb = 0 \\ &\Longrightarrow \quad A^TAx = A^Tb \qquad (1.5)\end{aligned}$$

假设 A 为满秩，$A^{T}A$为 n 阶对称正定矩阵，我们可以求得 x 的值为以下的形式：

$$x = (A^TA)^{-1}A^Tb \qquad (1.6)$$

1.3 非线性最小二乘问题

假设在（1.1）中，$f{i}(x)$为非线性函数，且 $F(x)$ 有连续偏导数。由于$f{i}(x)$为非线性函数，所以（1.2）中的非线性最小二乘无法套用（1.6）中的公式求得。 解这类问题的基本思想是，通过解一系列线性最小二乘问题求非线性最小二乘问题的解。设$x^{(k)}$是解的第 k 次近似。在$x^{(k)}$时，将函数$f_{i}(x)$线性化，从而将非线性最小二乘转换为线性最小二乘问题， 用（1.6）中的公式求解极小点$x^{(k+1)}$ ，把它作为非线性最小二乘问题解的第 $k+1$ 次近似。然后再从$x^{(k+1)}$出发，继续迭代。下面将来推导迭代公式。令

$$\begin{aligned}\varphi_i(x) &= f_i(x^{(k)}) + \nabla f_i(x^{(k)})^T(x - x^{(k)}) \\ &= \nabla f_i(x^{(k)})^Tx - [\nabla f_i(x^{(k)})^Tx^{(k)} - f_i(x^{(k)})] \quad ,i = 1,2,\ldots,m \qquad (1.7)\end{aligned}$$

上式右端是$f_{i}(x)$在$x^{(k)}$处展开的一阶泰勒级数多项式。令

$$\emptyset(\mathrm{x})=\sum_{i=1}^{m}\varphi_i^2(x) \qquad (1.8)$$

用 ø(x) 近似 F(x) ，从而用 ø(x) 的极小点作为目标函数 F(x) 的极小点的估计。现在求解线性最小二乘问题

$$\min \quad \emptyset(\mathrm{x}) \qquad (1.9)$$

把（1.9）写成

$$\emptyset(\mathrm{x})=(A_k\mathrm{x}-\mathrm{b})^T(A_kx-b) \qquad (1.10)$$

在公式（1.10）中，

$$A_k=\begin{bmatrix}\nabla f_1(x^{(k)})^T\\ \vdots\\ \nabla f_m(x^{(k)})^T\end{bmatrix}=\begin{bmatrix}\dfrac{\partial f_1(x^{(k)})}{\partial x_1} & \cdots & \dfrac{\partial f_1(x^{(k)})}{\partial x_n}\\ \vdots & \ddots & \vdots\\ \dfrac{\partial f_m(x^{(k)})}{\partial x_1} & \cdots & \dfrac{\partial f_m(x^{(k)})}{\partial x_n}\end{bmatrix}$$

$$\mathrm{b}=\begin{bmatrix}\nabla f_1(x^{(k)})^T x^{(k)}-f_1(x^{(k)})\\ \vdots\\ \nabla f_m(x^{(k)})^T x^{(k)}-f_m(x^{(k)})\end{bmatrix}=A_k x^{(k)}-f^{(k)}$$

将A_{k}和 b 带入公式（1.5）中，可以得到，

$$A_k^T A_k(\mathrm{x}-x^{(k)})=-A_k^T f^{(k)} \qquad (1.11)$$

如果$A{k}$为列满秩，且$A{k}^{T}A_{k}$是对称正定矩阵，那么由（1.11）可以得到 x 的极小值。

$$x^{(k+1)}=x^{(k)}-(A_k^T A_k)^{-1}A_k^T f^{(k)} \qquad (1.12)$$

可以推导出$2A{k}^{T}f{k}$是目标函数 F(x) 在$x^{(k)}$处的梯度，$2A{k}^{T}A{k}$是函数 ø(x) 的海森矩阵。所以（1.12）又可以写为如下形式。

$$x^{(k+1)}=x^{(k)}-H_k{}^{-1}\nabla\mathrm{F}(\mathrm{x}^{(k)}) \qquad (1.13)$$

公式（1.13）称为 `Gauss-Newton` 公式。向量

$$d^{(k)} = -(A_k^T A_k)^{-1} A_k^T f^{(k)} \qquad (1.14)$$

称为点$x^{(k)}$处的 `Gauss-Newton` 方向。为保证每次迭代能使目标函数值下降（至少不能上升），在求出$d^{(k)}$后，不直接使用$x^{(k)}+d^{(k)}$作为k+1次近似，而是从$x^{(k)}$出发，沿$d^{(k)}$方向进行一维搜索。

$$\min_{\lambda} \ F\left(x^{(k)} + \lambda d^{(k)}\right) \qquad (1.15)$$

求出步长$\lambda ^{(k)}$后，令

$$x^{(k+1)} = x^{(k)} - \lambda d^{(k)} \qquad (1.16)$$

最小二乘的计算步骤如下:

- （1） 给定初始点$x^{(1)}$，允许误差 `ε>0，k=1`
- （2） 计算函数值$f{i}(x)$，得到向量$f^{(k)}$，再计算一阶偏导，得到 `m*n` 矩阵$A{(k)}$
- （3） 解方程组（1.14）求得 `Gauss-Newton` 方向$d^{(k)}$
- （4）从$x^{(k)}$出发，沿着$d^{(k)}$作一维搜索，求出步长$\lambda ^{(k)}$，并令$x^{(k+1)}=x^{(k)}-\lambda d^{(k)}$
- （5）若$||x^{(k+1)}-x^{(k)}||<=\epsilon$停止迭代，求出 `x` ，否则， `k=k+1` ，返回步骤（2）

在某些情况下，矩阵$A^{T}A$是奇异的，这种情况下，我们无法求出它的逆矩阵，因此我们需要对其进行修改。用到的基本技巧是将一个正定对角矩阵添加到$A^{T}A$上，改变原来矩阵的特征值结构，使其变成条件较好的对称正定矩阵。典型的算法是 `Marquardt` 。

$$d^{(k)} = -(A_k^T A_k + \alpha_k I)^{-1} A_k^T f^{(k)} \qquad (1.17)$$

其中， `I` 是 `n` 阶单位矩阵， `alpha` 是一个正实数。当 `alpha` 为0时，$d^{(k)}$就是 `Gauss-Newton` 方向，当 `alpha` 充分大时，这时$d^{(k)}$接近 `F(x)` 在$x^{(k)}$处的最速下降方向。算法的具体过程见参考文献【1】。

2 共轭梯度法

2.1 共轭方向

在讲解共轭梯度法之前，我们需要先知道什么是共轭方向，下面的定义给出了答案。

定义**2.1** 设 `A` 是 `n*n` 对称正定矩阵，若两个方向$d^{(1)}$ 和$d^{(2)}$满足

$$d^{(1)T}Ad^{(2)} = 0 \qquad (2.1)$$

则称这两个方向关于 `A` 共轭。若$d^{(1)},d^{(2)},...,d^{(k)}$是 `k` 个方向，它们两两关于 `A` 共轭，则称这组方向是关于 `A` 共轭的。即

$$d^{(i)T}Ad^{(j)} = 0 \,, \mathrm{i} \neq \mathrm{j}; \mathrm{i}, \mathrm{j} = 1, \dots, \mathrm{k} \qquad (2.2)$$

在上述定义中，如果 `A` 是单位矩阵，那么两个方向关于 `A` 共轭等价于两个方向正交。如果 `A` 是一般的对称正定矩阵，$d^{(i)}$与$d^{(j)}$共轭，就是$d^{(i)}$与$Ad^{(j)}$正交。共轭方向有一些重要的性质。

定理**2.1** 设 `A` 是 `n` 阶对称正定矩阵，$d^{(1)},d^{(2)},...,d^{(k)}$是 `k` 个 `A` 的共轭的非零向量，则这个向量组线性无关。

定理**2.2 (**扩张子空间定理） 设有函数

$$\mathrm{f(x)} = \frac{1}{2}x^TAx + b^Tx + c$$

其中， `A` 是 `n` 阶对称正定矩阵，$d^{(1)},d^{(2)},...,d^{(k)}$是 `k` 个 `A` 的共轭的非零向量，以任意的$x^{(1)}$为初始点， 沿$d^{(1)},d^{(2)},...,d^{(k)}$进行一维搜索，得到$x^{(2)},x^{(3)},...,x^{(k+1)}$，则$d^{(k+1)}$是线性流型$x^{(1)}+H_{k}$上的唯一极小点，特别的，当k=n时，$x^{(n+1)}$是函数 `f(x)` 的唯一极小点。其中，

$$H_k = \left\{ x \middle| x = \sum_{i=1}^{k} \lambda_i d^{(i)}, \lambda_i \in (-\infty, +\infty) \right\}$$

是$d^{(1)},d^{(2)},...,d^{(k)}$生成的子空间。

这两个定理在文献【1】中有详细证明。

2.2 共轭梯度法

共轭梯度法的基本思想是把共轭性与最速下降方法相结合，利用已知点处的梯度构造一组共轭方向，并沿这组方向进行搜索，求出目标函数的极小点。这里我们仅仅讨论二次凸函数的共轭梯度法。

考虑问题

$$\min \quad \mathrm{f(x)} = \frac{1}{2}x^T Ax + b^T x + c \qquad (2.3)$$

其中 `A` 是对称正定矩阵， `c` 是常数。

具体求解方式如下：

首先给定任何一个初始点$x^{(1)}$ ，计算目标函数 `f(x)` 在这点的梯度$g{(1)}$ ，若$||g{(1)}||=0$ ，则停止计算；否则令

$$d^{(1)} = -\nabla f\left(x^{(1)}\right) = -g_1 \qquad (2.4)$$

沿方向$d^{(1)}$搜索，得到点$x^{(2)}$ 。计算$x^{(2)}$处的梯度，若$||g{(2)}||!=0$，则利用$g{(2)}$和$d^{(1)}$构造第二个搜索方向$d^{(2)}$，再沿$d^{(2)}$搜索。

一般的，若已知$x^{(k)}$和搜索方向$d^{(k)}$，则从$x^{(k))}$出发，沿方向$d^{(k)}$搜索，得到

$$x^{(\mathrm{k+1})} = x^{(\mathrm{k})} + \lambda_k d^{(k)} \qquad (2.5)$$

其中步长 `lambda` 满足

$$\mathrm{f}\left(x^{(\mathrm{k})} + \lambda_k d^{(k)}\right) = \min \ f\left(x^{(\mathrm{k})} + \lambda d^{(k)}\right)$$

此时可以求得 `lambda` 的显式表达。令

$$\varphi(\lambda) = f(x^{(k)} + \lambda_k d^{(k)})$$

通过求导可以求上面公式的极小值，即

$$\varphi'^{(\lambda)} = \nabla f(x^{(k+1)})^T d^{(k)} = 0 \qquad (2.6)$$

根据二次函数梯度表达式，（2.6）式可以推出如下方程

$$(Ax^{(k+1)} + b)^T d^{(k)} = 0 \ \Rightarrow \ (A(x^{(k)} + \lambda_k d^{(k)}) + b)^T d^{(k)} = 0$$
$$\Rightarrow (g_k + \lambda_k A d^{(k)})^T d^{(k)} = 0 \qquad (2.7)$$

由（2.7）式可以得到

$$\lambda_k = -\frac{g_k^T d^{(k)}}{d^{(k)T} A d^{(k)}} \qquad (2.8)$$

计算 `f(x)` 在$x^{(k+1)}$处的梯度，若$||g_{k+1}||=0$ ，则停止计算， 否则用$g_{(k+1)}$和$d^{(k)}$构造下一个搜索方向$d^{(k+1)}$ ，并使$d^{(k)}$与$d^{(k+1)}$共轭。按照这种设想，令

$$d^{(k+1)} = -g_{k+1} + \beta_k d^{(k)} \qquad (2.9)$$

在公式（2.9）两端同时乘以$d^{(k)T}A$，并令

$$d^{(k)T} A d^{(k+1)} = -d^{(k)T} A g_{k+1} + \beta_k d^{(k)T} A d^{(k)} = 0 \qquad (2.10)$$

可以求得

$$\beta_k = \frac{d^{(k)T} A g_{k+1}}{d^{(k)T} A d^{(k)}} \qquad (2.11)$$

再从$x^{(k+1)}$出发，沿$d^{(k+1)}$方向搜索。综上分析 ，在第1个搜索方向取负梯度的前提下，重复使用公式（2.5）、（2.8）、（2.9）、（2.11），我们就能够构造一组搜索方向。当然，前提是这组方向是关于 `A` 共轭的。 定理2.3说明了这组方向是共轭的。

定理**2.3** 对于正定二次函数（2.3），具有精确一维搜索的的共轭梯度法在 `m<=n` 次一维搜索后终止，并且对于所有 `i(1<=i<=m)` ，下列关系成立：

（1） $d^{(i)T}Ad^{(j)}=0,(j=1,2,..,i-1)$
（2） $g_i^T g_j=0,(j=1,2,..,i-1)$
（3） $g_i^T d^{(i)}=-g_i^T g_i\ ,(d^{(i)}\neq 0)$

还可以证明，对于正定二次函数，运用共轭梯度法时，不做矩阵运算也可以计算变量 `beta_k` 。

定理**2.4** 对于正定二次函数，共轭梯度法中因子 `beta_k` 具有下列表达式

$$\beta_k=\frac{\| g_{k+1}\|^2}{\| g_k\|^2},k\geq 1;g_k\neq 0 \qquad (2.12)$$

对于二次凸函数，共轭梯度法的计算步骤如下：

（1） 给定初始点$x^{(1)}$，设 k=1;
（2） 计算$g_k=\nabla f(x^{(k)})$，若$\| g_k\|=0$，则停止计算，得出极小值点。否则进行下一步；
（3） 构造搜索方向，令

$$d^{(k)}=-g_k+\beta_{k-1}d^{(k-1)}$$

当 k=1 时，$\beta_{k-1}=0$，当 k>0 时，$\beta_{k-1}=\frac{\|g_k\|^2}{\|g_{k-1}\|^2}$ ；

（4） 令$x^{(k+1)}=x^{(k)}+\lambda_k d^{(k)}$，按照$\lambda_k=-\frac{g_k^T d^{(k)}}{d^{(k)T}Ad^{(k)}}$计算步长$\lambda_k$;
（5） 若 k=n，则停止计算，得出极小值点。否则 k=k+2，返回步骤（2）。

3 最小二乘法在spark中的具体实现

`Spark ml` 中解决最小二乘可以选择两种方式，一种是非负正则化最小二乘，一种是乔里斯基分解（ `Cholesky` ）。

乔里斯基分解分解是把一个对称正定的矩阵表示成一个上三角矩阵 `U` 的转置和其本身的乘积的分解。在 `ml` 代码中，直接调用netlib-java封装的 `dppsv` 方法实现。

```
lapack.dppsv(“u”, k, 1, ne.ata, ne.atb, k, info)
```

可以深入 `dppsv` 代码（ `Fortran` 代码）了解更深的细节。我们分析的重点是非负正则化最小二乘的实现，因为在某些情况下，方程组的解为负数是没有意义的。虽然方程组可以得到精确解，但却不能取负值解。在这种情况下，其非负最小二乘解比方程的精确解更有意义。非负最小二乘问题要求解的问题如下公式

$$\min_x \ \frac{1}{2}x^T \mathrm{ata} x^T + x^T \mathrm{atb} \ , \quad x \geq 0 \qquad (3.1)$$

其中ata是半正定矩阵。

在 `ml` 代码中， `org.apache.spark.mllib.optimization.NNLS` 对象实现了非负最小二乘算法。该算法结合了投影梯度算法和共轭梯度算法来求解。

首先介绍 `NNLS` 对象中的 `Workspace` 类。

```
class Workspace(val n: Int) {
  val scratch = new Array[Double](n)
  val grad = new Array[Double](n)  //投影梯度
  val x = new Array[Double](n)
  val dir = new Array[Double](n)   //搜索方向
  val lastDir = new Array[Double](n)
  val res = new Array[Double](n)  //梯度
}
```

在 `Workspace` 中， `res` 表示梯度， `grad` 表示梯度的投影， `dir` 表示迭代过程中的搜索方向（共轭梯度中的搜索方向$d^{(k)}$）， `scratch` 代表公式（2.8）中的 $d^{(k)T}A$。

`NNLS` 对象中， `sort` 方法用来解最小二乘，它通过迭代求解极小值。我们将分步骤剖析该方法。

- （1）确定迭代次数。

```
val iterMax = math.max(400, 20 * n)
```

- （2）求梯度。

在每次迭代内部，第一步会求梯度 `res` ，代码如下

```
//y := alpha*A*x + beta*y 即  y:=1.0 * ata * x + 0.0 * res
blas.dgemv("N", n, n, 1.0, ata, n, x, 1, 0.0, res, 1)
// ata*x - atb
blas.daxpy(n, -1.0, atb, 1, res, 1)
```

`dgemv` 方法的作用是得到 `y := alpha*A*x + beta*y` ，在本代码中表示 `res=ata*x` 。 `daxpy` 方法的作用是得到 `y:=step*x +y` ,在本代码中表示 `res=ata*x-atb` ，即梯度。

- （**3**）求梯度的投影矩阵

求梯度矩阵的投影矩阵的依据如下公式。

$$p[\mathrm{grad}_i] = \begin{cases} 0, & if\ x_i = 0 \\ \mathrm{grad}_i\,, & if\ x_i > 0 \end{cases} \qquad (3.2)$$

详细代码如下所示：

```
//转换为投影矩阵
i = 0
while (i < n) {
   if (grad(i) > 0.0 && x(i) == 0.0) {
     grad(i) = 0.0
   }
   i = i + 1
}
```

- （**4**）求搜索方向。

在第一次迭代中，搜索方向即为梯度方向。如下面代码所示。

```
//在第一次迭代中，搜索方向dir即为梯度方向
blas.dcopy(n, grad, 1, dir, 1)
```

在第 `k` 次迭代中，搜索方向由梯度方向和前一步的搜索方向共同确定，计算依赖的公式是（2.9）。具体代码有两行

```
val alpha = ngrad / lastNorm
//alpha*lastDir + dir，此时dir为梯度方向
blas.daxpy(n, alpha, lastDir, 1, dir, 1)
```

此处的 `alpha` 就是根据公式（2.12）计算的。

- （**5**）计算步长。

知道了搜索方向，我们就可以依据公式（2.8）来计算步长。

```
def steplen(dir: Array[Double], res: Array[Double]): Double = {
  //top = g * d
val top = blas.ddot(n, dir, 1, res, 1)
  // y := alpha*A*x + beta*y.
  // scratch = d * ata
blas.dgemv("N", n, n, 1.0, ata, n, dir, 1, 0.0, scratch, 1)
  //公式（2.8），添加1e-20是为了避免分母为0
//g*d/d*ata*d
  top / (blas.ddot(n, scratch, 1, dir, 1) + 1e-20)
}
```

- （**6**）调整步长并修改迭代值。

因为解是非负的，所以步长需要做一定的处理，如果步长与搜索方向的乘积大于 `x` 的值，那么重置步长。重置逻辑如下：

```
i = 0
while (i < n) {
   if (step * dir(i) > x(i)) {
     //如果步长过大，那么二者的商替代
     step = x(i) / dir(i)
   }
   i = i + 1
}
```

最后，修改 `x` 的值，完成该次迭代。

```
i = 0
while (i < n) {
// x(i)趋向为0
  if (step * dir(i) > x(i) * (1 - 1e-14)) {
    x(i) = 0
    lastWall = iterno
  } else {
    x(i) -= step * dir(i)
  }
  i = i + 1
}
```

4 参考文献

【1】陈宝林，最优化理论和算法

【2】Boris T. Polyak，The conjugate gradient method in extreme problems

【3】https://en.wikipedia.org/wiki/Non-negative_least_squares

带权最小二乘

1 原理

给定n个带权的观察样本(w_i,a_i,b_i):

- w_i表示第i个观察样本的权重；
- a_i表示第i个观察样本的特征向量；
- b_i表示第i个观察样本的标签。

每个观察样本的特征数是m。我们使用下面的带权最小二乘公式作为目标函数：

$$minimize_{x}\frac{1}{2} \sum_{i=1}^n \frac{w_i(a_i^T x -b_i)^2}{\sum_{k=1}^n w_k} + \frac{1}{2}\frac{\lambda}{\delta}\sum_{j=1}^m(\sigma_{j} x_{j})^2$$

其中λ是正则化参数，δ是标签的总体标准差，σ_j是第j个特征列的总体标准差。

这个目标函数有一个解析解法，它仅仅需要一次处理样本来搜集必要的统计数据去求解。与原始数据集必须存储在分布式系统上不同， 如果特征数相对较小，这些统计数据可以加载进单机的内存中，然后在 `driver` 端使用乔里斯基分解求解目标函数。

`spark ml` 中使用 `WeightedLeastSquares` 求解带权最小二乘问题。 `WeightedLeastSquares` 仅仅支持 `L2` 正则化，并且提供了正则化和标准化的开关。为了使正太方程（ `normal equation` ）方法有效，特征数不能超过4096。如果超过4096，用 `L-BFGS` 代替。下面从代码层面介绍带权最小二乘优化算法 的实现。

2 代码解析

我们首先看看 `WeightedLeastSquares` 的参数及其含义。

```
private[ml] class WeightedLeastSquares(
    val fitIntercept: Boolean,  //是否使用截距
    val regParam: Double,    //L2正则化参数，指上面公式中的lambda
    val elasticNetParam: Double,  // alpha，控制L1和L2正则化
    val standardizeFeatures: Boolean, // 是否标准化特征
    val standardizeLabel: Boolean,  // 是否标准化标签
    val solverType: WeightedLeastSquares.Solver = WeightedLeastS
quares.Auto,
    val maxIter: Int = 100, // 迭代次数
    val tol: Double = 1e-6) extends Logging with Serializable

 sealed trait Solver
 case object Auto extends Solver
 case object Cholesky extends Solver
 case object QuasiNewton extends Solver
```

在上面的代码中，`standardizeFeatures` 决定是否标准化特征，如果为真，则σ_j是A第j个特征列的总体标准差，否则σ_j为1。`standardizeLabel` 决定是否标准化标签，如果为真，则δ是标签b的总体标准差，否则δ为1。`solverType` 指定求解的类型，有 `Auto`，`Cholesky` 和 `QuasiNewton` 三种选择。`tol` 表示迭代的收敛阈值，仅仅在 `solverType` 为 `QuasiNewton` 时可用。

2.1 求解过程

`WeightedLeastSquares` 接收一个包含（标签，权重，特征）的 `RDD`，使用 `fit` 方法训练，并返回 `WeightedLeastSquaresModel`。

```
def fit(instances: RDD[Instance]): WeightedLeastSquaresModel
```

训练过程分为下面几步。

- **1** 统计样本信息

```
val summary = instances.treeAggregate(new Aggregator)(_.add(_),
_.merge(_))
```

使用 `treeAggregate` 方法来统计样本信息。统计的信息在 `Aggregator` 类中给出了定义。通过展开上面的目标函数，我们可以知道这些统计信息的含义。

```
private class Aggregator extends Serializable {
    var initialized: Boolean = false
    var k: Int = _  // 特征数
    var count: Long = _  // 样本数
    var triK: Int = _  // 对角矩阵保存的元素个数
    var wSum: Double = _  // 权重和
    private var wwSum: Double = _  // 权重的平方和
    private var bSum: Double = _  // 带权标签和
    private var bbSum: Double = _  // 带权标签的平方和
    private var aSum: DenseVector = _  // 带权特征和
    private var abSum: DenseVector = _  // 带权特征标签相乘和
    private var aaSum: DenseVector = _  // 带权特征平方和
    }
```

方法 `add` 添加样本的统计信息，方法 `merge` 合并不同分区的统计信息。代码很简单，如下所示：

```
   /**
     * Adds an instance.
     */
    def add(instance: Instance): this.type = {
      val Instance(l, w, f) = instance
      val ak = f.size
      if (!initialized) {
        init(ak)
      }
      assert(ak == k, s"Dimension mismatch. Expect vectors of si
ze $k but got $ak.")
      count += 1L
      wSum += w
      wwSum += w * w
      bSum += w * l
      bbSum += w * l * l
      BLAS.axpy(w, f, aSum)
      BLAS.axpy(w * l, f, abSum)
      BLAS.spr(w, f, aaSum) // wff^T
```

```
      this
    }

   /**
     * Merges another [[Aggregator]].
     */
    def merge(other: Aggregator): this.type = {
      if (!other.initialized) {
        this
      } else {
        if (!initialized) {
          init(other.k)
        }
        assert(k == other.k, s"dimension mismatch: this.k = $k b
ut other.k = ${other.k}")
        count += other.count
        wSum += other.wSum
        wwSum += other.wwSum
        bSum += other.bSum
        bbSum += other.bbSum
        BLAS.axpy(1.0, other.aSum, aSum)
        BLAS.axpy(1.0, other.abSum, abSum)
        BLAS.axpy(1.0, other.aaSum, aaSum)
        this
      }
```

`Aggregator` 类给出了以下一些统计信息：

```
aBar: 特征加权平均数
bBar: 标签加权平均数
aaBar: 特征平方加权平均数
bbBar: 标签平方加权平均数
aStd: 特征的加权总体标准差
bStd: 标签的加权总体标准差
aVar: 带权的特征总体方差
```

计算出这些信息之后，将均值缩放到标准空间，即使每列数据的方差为1。

```
// 缩放bBar和 bbBar
```

```
val bBar = summary.bBar / bStd
val bbBar = summary.bbBar / (bStd * bStd)

val aStd = summary.aStd
val aStdValues = aStd.values
// 缩放aBar
val aBar = {
      val _aBar = summary.aBar
      val _aBarValues = _aBar.values
      var i = 0
      // scale aBar to standardized space in-place
      while (i < numFeatures) {
        if (aStdValues(i) == 0.0) {
          _aBarValues(i) = 0.0
        } else {
          _aBarValues(i) /= aStdValues(i)
        }
        i += 1
      }
      _aBar
}
val aBarValues = aBar.values
// 缩放 abBar
val abBar = {
      val _abBar = summary.abBar
      val _abBarValues = _abBar.values
      var i = 0
      // scale abBar to standardized space in-place
      while (i < numFeatures) {
        if (aStdValues(i) == 0.0) {
          _abBarValues(i) = 0.0
        } else {
          _abBarValues(i) /= (aStdValues(i) * bStd)
        }
        i += 1
      }
      _abBar
}
val abBarValues = abBar.values
// 缩放aaBar
```

```
val aaBar = {
      val _aaBar = summary.aaBar
      val _aaBarValues = _aaBar.values
      var j = 0
      var p = 0
      // scale aaBar to standardized space in-place
      while (j < numFeatures) {
        val aStdJ = aStdValues(j)
        var i = 0
        while (i <= j) {
          val aStdI = aStdValues(i)
          if (aStdJ == 0.0 || aStdI == 0.0) {
            _aaBarValues(p) = 0.0
          } else {
            _aaBarValues(p) /= (aStdI * aStdJ)
          }
          p += 1
          i += 1
        }
        j += 1
      }
      _aaBar
}
val aaBarValues = aaBar.values
```

- **2** 处理**L2**正则项

```
val effectiveRegParam = regParam / bStd
val effectiveL1RegParam = elasticNetParam * effectiveRegParam
val effectiveL2RegParam = (1.0 - elasticNetParam) * effectiveReg
Param

// 添加L2正则项到对角矩阵中
var i = 0
var j = 2
while (i < triK) {
   var lambda = effectiveL2RegParam
   if (!standardizeFeatures) {
       val std = aStdValues(j - 2)
       if (std != 0.0) {
          lambda /= (std * std) //正则项标准化
       } else {
          lambda = 0.0
       }
   }
   if (!standardizeLabel) {
        lambda *= bStd
   }
   aaBarValues(i) += lambda
   i += j
   j += 1
}
```

- **3 选择solver**

`WeightedLeastSquares` 实现了 `CholeskySolver` 和 `QuasiNewtonSolver` 两种不同的求解方法。当没有正则化项时， 选择 `CholeskySolver` 求解，否则用 `QuasiNewtonSolver` 求解。

```
val solver = if ((solverType == WeightedLeastSquares.Auto && ela
sticNetParam != 0.0 &&
      regParam != 0.0) || (solverType == WeightedLeastSquares.Qu
asiNewton)) {
      val effectiveL1RegFun: Option[(Int) => Double] = if (effec
tiveL1RegParam != 0.0) {
        Some((index: Int) => {
            if (fitIntercept && index == numFeatures) {
              0.0
            } else {
              if (standardizeFeatures) {
                effectiveL1RegParam
              } else {
                if (aStdValues(index) != 0.0) effectiveL1RegPara
m / aStdValues(index) else 0.0
              }
            }
          })
      } else {
        None
      }
      new QuasiNewtonSolver(fitIntercept, maxIter, tol, effectiv
eL1RegFun)
    } else {
      new CholeskySolver
    }
```

`CholeskySolver` 和 `QuasiNewtonSolver` 的详细分析会在另外的专题进行描述。

- **4** 处理结果

```
val solution = solver match {
      case cholesky: CholeskySolver =>
        try {
          cholesky.solve(bBar, bbBar, ab, aa, aBar)
        } catch {
          // if Auto solver is used and Cholesky fails due to si
ngular AtA, then fall back to
          // Quasi-Newton solver.
          case _: SingularMatrixException if solverType == Weigh
tedLeastSquares.Auto =>
            logWarning("Cholesky solver failed due to singular c
ovariance matrix. " +
              "Retrying with Quasi-Newton solver.")
            // ab and aa were modified in place, so reconstruct
them
            val _aa = getAtA(aaBarValues, aBarValues)
            val _ab = getAtB(abBarValues, bBar)
            val newSolver = new QuasiNewtonSolver(fitIntercept,
maxIter, tol, None)
            newSolver.solve(bBar, bbBar, _ab, _aa, aBar)
        }
      case qn: QuasiNewtonSolver =>
        qn.solve(bBar, bbBar, ab, aa, aBar)
    }

    val (coefficientArray, intercept) = if (fitIntercept) {
      (solution.coefficients.slice(0, solution.coefficients.leng
th - 1),
        solution.coefficients.last * bStd)
    } else {
      (solution.coefficients, 0.0)
    }
```

上面代码的异常处理需要注意一下。在 `AtA` 是奇异矩阵的情况下，乔里斯基分解会报错，这时需要用拟牛顿方法求解。

以上的结果是在标准空间中，所以我们需要将结果从标准空间转换到原来的空间。

```
// convert the coefficients from the scaled space to the origina
l space
var q = 0
val len = coefficientArray.length
while (q < len) {
   coefficientArray(q) *= { if (aStdValues(q) != 0.0) bStd / aSt
dValues(q) else 0.0 }
   q += 1
}
```

奇异值分解

1 奇异值分解

在了解特征值分解之后，我们知道，矩阵 `A` 不一定是方阵。为了得到方阵，可以将矩阵 `A` 的转置乘以该矩阵。从而可以得到公式：

$$(A^TA)v = \lambda v$$

现在假设存在 `M*N` 矩阵 `A` ，我们的目标是在 `n` 维空间中找一组正交基，使得经过 `A` 变换后还是正交的。假设已经找到这样一组正交基：

$$\{v_1, v_2, \dots, v_n\}$$

`A` 矩阵可以将这组正则基映射为如下的形式。

$$\{Av_1, Av_2, \dots, Av_n\}$$

要使上面的基也为正则基，即使它们两两正交，那么需要满足下面的条件。

$$Av_i.Av_j = (Av_i)^TAv_j = v_i^TA^TAv_j = 0$$

如果正交基 `v` 选择为$A^{T}A$的特征向量的话，由于$A^{T}A$是对称阵， `v` 之间两两正交，那么

$$v_i^TA^TAv_j = v_i^T\lambda_jv_j = \lambda_jv_i^Tv_j = \lambda_jv_i.v_j = 0$$

由于下面的公式成立

$$|Av_i|^2 = Av_i.Av_i = \lambda_iv_i.v_i = \lambda_i \geq 0$$

所以取单位向量

$$u_i = \frac{Av_i}{|Av_i|} = \frac{1}{\sqrt{\lambda_i}} Av_i$$

可以得到

$$Av_i = \sigma_i u_i, \text{其中奇异值} \sigma_i = \frac{1}{\sqrt{\lambda_i}}$$

奇异值分解是一个能适用于任意的矩阵的一种分解的方法，它的形式如下：

$$A = U \Sigma V^T$$

其中， `U` 是一个 `M*M` 的方阵，它包含的向量是正交的，称为左奇异向量（即上文的 `u` ）。 `sigma` 是一个 `N*M` 的对角矩阵，每个对角线上的元素就是一个奇异值。 `V` 是一个 `N*N` 的矩阵，它包含的向量是正交的，称为右奇异向量（即上文的 `v` ）。

2 源码分析

`MLlib` 在 `RowMatrix` 类中实现了奇异值分解。下面是一个使用奇异值分解的例子。

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.distributed.RowMatrix
import org.apache.spark.mllib.linalg.SingularValueDecomposition
val mat: RowMatrix = ...
// Compute the top 20 singular values and corresponding singular
 vectors.
val svd: SingularValueDecomposition[RowMatrix, Matrix] = mat.com
puteSVD(20, computeU = true)
val U: RowMatrix = svd.U // The U factor is a RowMatrix.
val s: Vector = svd.s // The singular values are stored in a loc
al dense vector.
val V: Matrix = svd.V // The V factor is a local dense matrix.
```

2.1 性能

我们假设 `n` 比 `m` 小。奇异值和右奇异值向量可以通过方阵$A^{T}A$的特征值和特征向量得到。左奇异向量通过AVS^{-1}求得。 `ml` 实际使用的方法方法依赖计算花费。

- 当 `n` 很小（ `n<100` ）或者 `k` 比 `n` 大(`k>n/2`)，我们会首先计算方阵$A^{T}A$ ，然后在 `driver` 本地计算它的 `top` 特征值和特征向量。它的空间复杂度是 `O(n*n)` ，时间复杂度是 `O(n*n*k)` 。
- 否则，我们用分布式的方式先计算$A^{T}Av$,然后把它传给ARPACK在 `driver` 上计算 `top` 特征值和特征向量。它需要传递 `O(k)` 的数据，每个 `executor` 的空间复杂度是 `O(n)` , `driver` 的空间复杂度是 `O(nk)`

2.2 代码实现

```
def computeSVD(
      k: Int,
      computeU: Boolean = false,
      rCond: Double = 1e-9): SingularValueDecomposition[RowMatrix
, Matrix] = {
    // 迭代次数
    val maxIter = math.max(300, k * 3)
    // 阈值
    val tol = 1e-10
    computeSVD(k, computeU, rCond, maxIter, tol, "auto")
}
```

`computeSVD(k, computeU, rCond, maxIter, tol, "auto")` 的实现分为三步。分别是选择计算模式，$A^{T}A$的特征值分解，计算 `V` , `U` , `Sigma` 。 下面分别介绍这三步。

- **1** 选择计算模式

```
val computeMode = mode match {
      case "auto" =>
        if (k > 5000) {
          logWarning(s"computing svd with k=$k and n=$n, please
check necessity")
        }
        if (n < 100 || (k > n / 2 && n <= 15000)) {
          // 满足上述条件，首先计算方阵，然后本地计算特征值，避免数据传递
          if (k < n / 3) {
            SVDMode.LocalARPACK
          } else {
            SVDMode.LocalLAPACK
          }
        } else {
          // 分布式实现
          SVDMode.DistARPACK
        }
      case "local-svd" => SVDMode.LocalLAPACK
      case "local-eigs" => SVDMode.LocalARPACK
      case "dist-eigs" => SVDMode.DistARPACK
}
```

- **2 特征值分解**

```
 val (sigmaSquares: BDV[Double], u: BDM[Double]) = computeMode m
atch {
      case SVDMode.LocalARPACK =>
        val G = computeGramianMatrix().toBreeze.asInstanceOf[BDM[
Double]]
        EigenValueDecomposition.symmetricEigs(v => G * v, n, k,
tol, maxIter)
      case SVDMode.LocalLAPACK =>
        // breeze (v0.10) svd latent constraint, 7 * n * n + 4 *
 n < Int.MaxValue
        val G = computeGramianMatrix().toBreeze.asInstanceOf[BDM[
Double]]
        val brzSvd.SVD(uFull: BDM[Double], sigmaSquaresFull: BDV[
Double], _) = brzSvd(G)
        (sigmaSquaresFull, uFull)
      case SVDMode.DistARPACK =>
        if (rows.getStorageLevel == StorageLevel.NONE) {
          logWarning("The input data is not directly cached, whi
ch may hurt performance if its"
            + " parent RDDs are also uncached.")
        }
        EigenValueDecomposition.symmetricEigs(multiplyGramianMat
rixBy, n, k, tol, maxIter)
    }
```

当计算模式是 `SVDMode.LocalARPACK` 和 `SVDMode.LocalLAPACK` 时，程序实现的步骤是先获取方阵$A^{T}A$ ，在计算其特征值和特征向量。 获取方阵无需赘述，我们只需要注意它无法处理列大于65535的矩阵。我们分别看这两种模式下，如何获取特征值和特征向量。

在 `SVDMode.LocalARPACK` 模式下，使用 `EigenValueDecomposition.symmetricEigs(v => G * v, n, k, tol, maxIter)` 计算特征值和特征向量。在 `SVDMode.LocalLAPACK` 模式下，直接使用 `breeze` 的方法计算。

在 `SVDMode.DistARPACK` 模式下，不需要先计算方阵，但是传入 `EigenValueDecomposition.symmetricEigs` 方法的函数不同。

```
 private[mllib] def multiplyGramianMatrixBy(v: BDV[Double]): BDV[
Double] = {
    val n = numCols().toInt
    //v作为广播变量
    val vbr = rows.context.broadcast(v)
    rows.treeAggregate(BDV.zeros[Double](n))(
      seqOp = (U, r) => {
        val rBrz = r.toBreeze
        val a = rBrz.dot(vbr.value)
        rBrz match {
          //计算y += x * a
          case _: BDV[_] => brzAxpy(a, rBrz.asInstanceOf[BDV[Dou
ble]], U)
          case _: BSV[_] => brzAxpy(a, rBrz.asInstanceOf[BSV[Dou
ble]], U)
          case _ => throw new UnsupportedOperationException
        }
        U
      }, combOp = (U1, U2) => U1 += U2)
  }
```

特征值分解的具体分析在特征值分解中有详细分析，请参考该文了解详情。

- **3** 计算 `U` , `V` 以及 `Sigma`

```
    //获取特征值向量
    val sigmas: BDV[Double] = brzSqrt(sigmaSquares)
    val sigma0 = sigmas(0)
    val threshold = rCond * sigma0
    var i = 0
    // sigmas的长度可能会小于k
    // 所以使用 i < min(k, sigmas.length) 代替 i < k.
    if (sigmas.length < k) {
      logWarning(s"Requested $k singular values but only found $
{sigmas.length} converged.")
    }
    while (i < math.min(k, sigmas.length) && sigmas(i) >= thresh
old) {
```

```
      i += 1
    }
    val sk = i
    if (sk < k) {
      logWarning(s"Requested $k singular values but only found $
sk nonzeros.")
    }
    //计算s，也即sigma
    val s = Vectors.dense(Arrays.copyOfRange(sigmas.data, 0, sk)
)
    //计算V
    val V = Matrices.dense(n, sk, Arrays.copyOfRange(u.data, 0,
n * sk))
    //计算U
    // N = Vk * Sk^{-1}
    val N = new BDM[Double](n, sk, Arrays.copyOfRange(u.data, 0,
 n * sk))
    var i = 0
    var j = 0
    while (j < sk) {
        i = 0
        val sigma = sigmas(j)
        while (i < n) {
          //对角矩阵的逆即为倒数
          N(i, j) /= sigma
          i += 1
        }
        j += 1
    }
    //U=A * N
    val U = this.multiply(Matrices.fromBreeze(N))
```

参考文献

【1】强大的矩阵奇异值分解(SVD)及其应用

【2】奇异值分解(SVD)原理详解及推导

【3】A Singularly Valuable Decomposition: The SVD of a Matrix

特征值分解

假设向量 `v` 是方阵 `A` 的特征向量，可以表示成下面的形式：

$$Av = \lambda v$$

这里 `lambda` 表示特征向量 `v` 所对应的特征值。并且一个矩阵的一组特征向量是一组正交向量。特征值分解是将一个矩阵分解为下面的形式：

$$A = Q\Sigma Q^{-1} = Q\Sigma Q^{T}$$

其中 `Q` 是这个矩阵 `A` 的特征向量组成的矩阵。 `sigma` 是一个对角矩阵，每个对角线上的元素就是一个特征值。

特征值分解是一个提取矩阵特征很不错的方法，但是它只适合于方阵，对于非方阵，它不适合。这就需要用到奇异值分解。

1 源码分析

`MLlib` 使用 `ARPACK` 来求解特征值分解。它的实现代码如下

```
def symmetricEigs(
      mul: BDV[Double] => BDV[Double],
      n: Int,
      k: Int,
      tol: Double,
      maxIterations: Int): (BDV[Double], BDM[Double]) = {
    val arpack = ARPACK.getInstance()
    // tolerance used in stopping criterion
    val tolW = new doubleW(tol)
    // number of desired eigenvalues, 0 < nev < n
    val nev = new intW(k)
    // nev Lanczos vectors are generated in the first iteration
    // ncv-nev Lanczos vectors are generated in each subsequent
iteration
    // ncv must be smaller than n
```

```
    val ncv = math.min(2 * k, n)
    // "I" for standard eigenvalue problem, "G" for generalized
eigenvalue problem
    val bmat = "I"
    // "LM" : compute the NEV largest (in magnitude) eigenvalues
    val which = "LM"
    var iparam = new Array[Int](11)
    // use exact shift in each iteration
    iparam(0) = 1
    // maximum number of Arnoldi update iterations, or the actua
l number of iterations on output
    iparam(2) = maxIterations
    // Mode 1: A*x = lambda*x, A symmetric
    iparam(6) = 1

    var ido = new intW(0)
    var info = new intW(0)
    var resid = new Array[Double](n)
    var v = new Array[Double](n * ncv)
    var workd = new Array[Double](n * 3)
    var workl = new Array[Double](ncv * (ncv + 8))
    var ipntr = new Array[Int](11)

    // call ARPACK's reverse communication, first iteration with
 ido = 0
    arpack.dsaupd(ido, bmat, n, which, nev.`val`, tolW, resid, n
cv, v, n, iparam, ipntr, workd,
      workl, workl.length, info)
    val w = BDV(workd)
    // ido = 99 : done flag in reverse communication
    while (ido.`val` != 99) {
      if (ido.`val` != -1 && ido.`val` != 1) {
        throw new IllegalStateException("ARPACK returns ido = "
+ ido.`val` +
            " This flag is not compatible with Mode 1: A*x = lam
bda*x, A symmetric.")
      }
      // multiply working vector with the matrix
      val inputOffset = ipntr(0) - 1
      val outputOffset = ipntr(1) - 1
```

```
      val x = w.slice(inputOffset, inputOffset + n)
      val y = w.slice(outputOffset, outputOffset + n)
      y := mul(x)
      // call ARPACK's reverse communication
      arpack.dsaupd(ido, bmat, n, which, nev.`val`, tolW, resid,
 ncv, v, n, iparam, ipntr,
        workd, workl, workl.length, info)
    }

    val d = new Array[Double](nev.`val`)
    val select = new Array[Boolean](ncv)
    // copy the Ritz vectors
    val z = java.util.Arrays.copyOfRange(v, 0, nev.`val` * n)

    // call ARPACK's post-processing for eigenvectors
    arpack.dseupd(true, "A", select, d, z, n, 0.0, bmat, n, whic
h, nev, tol, resid, ncv, v, n,
      iparam, ipntr, workd, workl, workl.length, info)

    // number of computed eigenvalues, might be smaller than k
    val computed = iparam(4)

    val eigenPairs = java.util.Arrays.copyOfRange(d, 0, computed
).zipWithIndex.map { r =>
      (r._1, java.util.Arrays.copyOfRange(z, r._2 * n, r._2 * n
+ n))
    }

    // sort the eigen-pairs in descending order
    val sortedEigenPairs = eigenPairs.sortBy(- _._1)

    // copy eigenvectors in descending order of eigenvalues
    val sortedU = BDM.zeros[Double](n, computed)
    sortedEigenPairs.zipWithIndex.foreach { r =>
      val b = r._2 * n
      var i = 0
      while (i < n) {
        sortedU.data(b + i) = r._1._2(i)
        i += 1
      }
```

```
    }
    (BDV[Double](sortedEigenPairs.map(_._1)), sortedU)
  }
```

我们可以查看 `ARPACK` 的注释详细了解 `dsaupd` 和 `dseupd` 方法的作用。

主成分分析

1 主成分分析原理

主成分分析是最常用的一种降维方法。我们首先考虑一个问题：对于正交矩阵空间中的样本点，如何用一个超平面对所有样本进行恰当的表达。容易想到，如果这样的超平面存在，那么他大概应该具有下面的性质。

- 最近重构性：样本点到超平面的距离都足够近
- 最大可分性：样本点在这个超平面上的投影尽可能分开

基于最近重构性和最大可分性，能分别得到主成分分析的两种等价推导。

1.1 最近重构性

假设我们对样本点进行了中心化，即所有样本的和为0。再假设投影变换后得到的新坐标系为：

$$\{w_1, w_2, \dots, w_d\}, \qquad ||w_i||_2 = 1, w_i{}^T w_j = 0$$

若丢弃新坐标系中的部分坐标，将维度降到 `d'` ，则样本点$x*{i}*$在低位坐标系中的投影是$z{i}$ ：

$$z_i = (z_{i1}; z_{i2}; \dots; z_{id'}), z_{ij} = w_j^T x_i$$

这里$z*{ij}*$是$x{i}$在低维坐标系下第 `j` 维的坐标。若基于$z*{i}*$来重构$x{i}$，那么可以得到

$$\overline{x}_i = \sum_{j=1}^{d'} z_{ij} w_j$$

考虑整个训练集，原样本点和基于投影重构的样本点之间的距离为

$$\sum_{i=1}^{m} \| \sum_{j=1}^{d'} z_{ij} w_j - x_i \|_2^2 = \sum_{i=1}^{m} z_i^T z_i - 2 \sum_{i=1}^{m} z_i^T W^T x_i + c$$

$$\propto -\mathrm{tr}(W^T \left(\sum_{i=1}^{m} x_i x_i^T \right) \mathrm{W})$$

根据最近重构性，最小化上面的式子，就可以得到主成分分析的优化目标

$$min_w \ -tr(W^T X X^T W), s.t. W^T W = I$$

1.2 最大可分性

从最大可分性出发，我们可以得到主成分分析的另一种解释。我们知道，样本点$x{i}$在新空间中超平面上的投影是$W^{T}x{i}$，若所有样本点的投影能尽可能分开，则应该使投影后样本点的方差最大化。投影后样本点的方差是

$$\sum_i W^T x_i x_i^T W$$

于是优化目标可以写为

$$max_w \ tr(W^T X X^T W), s.t. W^T W = I$$

这个优化目标和上文的优化目标是等价的。对优化目标使用拉格朗日乘子法可得

$$XX^T W = \lambda W$$

于是，只需要对协方差矩阵进行特征值分解，将得到的特征值排序，在取前 `d'` 个特征值对应的特征向量，即得到主成分分析的解。

2 源码分析

2.1 实例

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.distributed.RowMatrix
val mat: RowMatrix = ...
// Compute the top 10 principal components.
val pc: Matrix = mat.computePrincipalComponents(10) // Principal components are stored in a local dense matrix.
// Project the rows to the linear space spanned by the top 10 principal components.
val projected: RowMatrix = mat.multiply(pc)
```

2.2 实现代码

主成分分析的实现代码在 `RowMatrix` 中实现。源码如下：

```
def computePrincipalComponents(k: Int): Matrix = {
    val n = numCols().toInt
    //计算协方差矩阵
    val Cov = computeCovariance().toBreeze.asInstanceOf[BDM[Double]]
    //特征值分解
    val brzSvd.SVD(u: BDM[Double], _, _) = brzSvd(Cov)
    if (k == n) {
      Matrices.dense(n, k, u.data)
    } else {
      Matrices.dense(n, k, Arrays.copyOfRange(u.data, 0, n * k))
    }
  }
```

这段代码首先会计算样本的协方差矩阵，然后在通过 `breeze` 的 `svd` 方法进行奇异值分解。这里由于协方差矩阵是方阵，所以奇异值分解等价于特征值分解。下面是计算协方差的代码

```
 def computeCovariance(): Matrix = {
    val n = numCols().toInt
    checkNumColumns(n)
    val (m, mean) = rows.treeAggregate[(Long, BDV[Double])]((0L,
BDV.zeros[Double](n)))(
      seqOp = (s: (Long, BDV[Double]), v: Vector) => (s._1 + 1L,
 s._2 += v.toBreeze),
      combOp = (s1: (Long, BDV[Double]), s2: (Long, BDV[Double])
) =>
        (s1._1 + s2._1, s1._2 += s2._2)
    )
    updateNumRows(m)
    mean :/= m.toDouble
    // We use the formula Cov(X, Y) = E[X * Y] - E[X] E[Y], whic
h is not accurate if E[X * Y] is
    // large but Cov(X, Y) is small, but it is good for sparse c
omputation.
    // TODO: find a fast and stable way for sparse data.
    val G = computeGramianMatrix().toBreeze.asInstanceOf[BDM[Dou
ble]]
    var i = 0
    var j = 0
    val m1 = m - 1.0
    var alpha = 0.0
    while (i < n) {
      alpha = m / m1 * mean(i)
      j = i
      while (j < n) {
        val Gij = G(i, j) / m1 - alpha * mean(j)
        G(i, j) = Gij
        G(j, i) = Gij
        j += 1
      }
      i += 1
    }
    Matrices.fromBreeze(G)
  }
```

参考文献

【1】 机器学习.周志华

TF-IDF

1 介绍

词频-逆文档频率法(`Term frequency-inverse document frequency,TF-IDF`)是在文本挖掘中广泛使用的特征向量化方法。 它反映语料中词对文档的重要程度。假设用 `t` 表示词， `d` 表示文档， `D` 表示语料。词频 `TF(t,d)` 表示词 `t` 在文档 `d` 中出现的次数。文档频率 `DF(t,D)` 表示语料中出现词 `t` 的文档的个数。 如果我们仅仅用词频去衡量重要程度，这很容易过分强调出现频繁但携带较少文档信息的词，如 `of` 、 `the` 等。如果一个词在语料中出现很频繁，这意味着它不携带特定文档的特殊信息。逆文档频率数值衡量一个词提供多少信息。

$$\mathrm{IDF}(t, D) = \log\frac{|D| + 1}{DF(t, D) + 1}$$

如果某个词出现在所有的文档中，它的 `IDF` 值为0。注意，上式有个平滑项，这是为了避免分母为0的情况发生。 `TF-IDF` 就是 `TF` 和 `IDF` 简单的相乘。

$$\mathrm{TFIDF}(t, d, D) = \mathrm{TF}(t, d) * \mathrm{IDF}(t, D)$$

词频和文档频率的定义有很多种不同的变种。在 `Mllib` 中，分别提供了 `TF` 和 `IDF` 的实现，以便有更好的灵活性。

`Mllib` 使用 `hashing trick` 实现词频。元素的特征应用一个 `hash` 函数映射到一个索引（即词），通过这个索引计算词频。这个方法避免计算全局的词-索引映射，因为全局的词-索引映射在大规模语料中花费较大。 但是，它会出现哈希冲突，这是因为不同的元素特征可能得到相同的哈希值。为了减少碰撞冲突，我们可以增加目标特征的维度，例如哈希表的桶数量。默认的特征维度是1048576。

2 实例

- TF的计算

```
import org.apache.spark.rdd.RDD
import org.apache.spark.SparkContext
import org.apache.spark.mllib.feature.HashingTF
import org.apache.spark.mllib.linalg.Vector
val sc: SparkContext = ...
// Load documents (one per line).
val documents: RDD[Seq[String]] = sc.textFile("...").map(_.split(
" ").toSeq)
val hashingTF = new HashingTF()
val tf: RDD[Vector] = hashingTF.transform(documents)
```

- IDF的计算

```
import org.apache.spark.mllib.feature.IDF
// ... continue from the previous example
tf.cache()
val idf = new IDF().fit(tf)
val tfidf: RDD[Vector] = idf.transform(tf)
//或者
val idf = new IDF(minDocFreq = 2).fit(tf)
val tfidf: RDD[Vector] = idf.transform(tf)
```

3 源码实现

下面分别分析 `HashingTF` 和 `IDF` 的实现。

3.1 HashingTF

```
def transform(document: Iterable[_]): Vector = {
    val termFrequencies = mutable.HashMap.empty[Int, Double]
    document.foreach { term =>
      val i = indexOf(term)
      termFrequencies.put(i, termFrequencies.getOrElse(i, 0.0) +
1.0)
    }
    Vectors.sparse(numFeatures, termFrequencies.toSeq)
  }
```

以上代码中， `indexOf` 方法使用哈希获得索引。

```
//为了减少碰撞，将numFeatures设置为1048576
def indexOf(term: Any): Int = Utils.nonNegativeMod(term.##, numF
eatures)
def nonNegativeMod(x: Int, mod: Int): Int = {
    val rawMod = x % mod
    rawMod + (if (rawMod < 0) mod else 0)
  }
```

这里的 `term.##` 等价于 `term.hashCode` ，得到哈希值之后，作取余操作得到相应的索引。

3.2 IDF

我们先看 `IDF` 的 `fit` 方法。

```
def fit(dataset: RDD[Vector]): IDFModel = {
    val idf = dataset.treeAggregate(new IDF.DocumentFrequencyAgg
regator(
          minDocFreq = minDocFreq))(
      seqOp = (df, v) => df.add(v),
      combOp = (df1, df2) => df1.merge(df2)
    ).idf()
    new IDFModel(idf)
  }
```

该函数使用 `treeAggregate` 处理数据集，生成一个 `DocumentFrequencyAggregator` 对象，它用于计算文档频率。重点看 `add` 和 `merge` 方法。

```
def add(doc: Vector): this.type = {
      if (isEmpty) {
        df = BDV.zeros(doc.size)
      }
      //计算
      doc match {
        case SparseVector(size, indices, values) =>
          val nnz = indices.size
          var k = 0
          while (k < nnz) {
            if (values(k) > 0) {
              df(indices(k)) += 1L
            }
            k += 1
          }
        case DenseVector(values) =>
          val n = values.size
          var j = 0
          while (j < n) {
            if (values(j) > 0.0) {
              df(j) += 1L
            }
            j += 1
          }
        case other =>
          throw new UnsupportedOperationException
      }
      m += 1L
      this
    }
```

`df` 这个向量的每个元素都表示该索引对应的词出现的文档数。`m` 表示文档总数。

```
def merge(other: DocumentFrequencyAggregator): this.type = {
      if (!other.isEmpty) {
        m += other.m
        if (df == null) {
          df = other.df.copy
        } else {
          //简单的向量相加
          df += other.df
        }
      }
      this
    }
```

`treeAggregate` 方法处理完数据之后，调用 `idf` 方法将文档频率低于给定值的词的 `idf` 置为0，其它的按照上面的公式计算。

```
 def idf(): Vector = {
      val n = df.length
      val inv = new Array[Double](n)
      var j = 0
      while (j < n) {
        if (df(j) >= minDocFreq) {
          //计算得到idf
          inv(j) = math.log((m + 1.0) / (df(j) + 1.0))
        }
        j += 1
      }
      Vectors.dense(inv)
}
```

最后使用 `transform` 方法计算 `tfidf` 值。

```
  //这里的dataset指tf
  def transform(dataset: RDD[Vector]): RDD[Vector] = {
     val bcIdf = dataset.context.broadcast(idf)
     dataset.mapPartitions(iter => iter.map(v => IDFModel.transfo
rm(bcIdf.value, v)))
   }
  def transform(idf: Vector, v: Vector): Vector = {
      val n = v.size
      v match {
        case SparseVector(size, indices, values) =>
          val nnz = indices.size
          val newValues = new Array[Double](nnz)
          var k = 0
          while (k < nnz) {
            //tf-idf = tf * idf
            newValues(k) = values(k) * idf(indices(k))
            k += 1
          }
          Vectors.sparse(n, indices, newValues)
        case DenseVector(values) =>
          val newValues = new Array[Double](n)
          var j = 0
          while (j < n) {
            newValues(j) = values(j) * idf(j)
            j += 1
          }
          Vectors.dense(newValues)
        case other =>
          throw new UnsupportedOperationException
      }
    }
```

Word2Vector

Word2Vector将词转换成分布式向量。分布式表示的主要优势是相似的词在向量空间距离较近，这使我们更容易泛化新的模式并且使模型估计更加健壮。 分布式的向量表示在许多自然语言处理应用（如命名实体识别、消歧、词法分析、机器翻译）中非常有用。

1 模型

在 `MLlib` 中， `Word2Vector` 使用 `skip-gram` 模型来实现。 `skip-gram` 的训练目标是学习词向量表示，这个表示可以很好的预测它在相同句子中的上下文。数学上，给定训练词 `w_1,w_2,...,w_T` ， `skip-gram` 模型的目标是最大化下面的平均对数似然。

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{j=-k}^{j=k}\log p\left(w_{t+j}|w_t\right)$$

其中 `k` 是训练窗口的大小。在 `skip-gram` 模型中，每个词 `w` 和两个向量 `u_w` 和 `v_w` 相关联，这两个向量分别表示词和上下文。正确地预测给定词 `w_j` 的条件下 `w_i` 的概率使用 `softmax` 模型。

$$p(w_i|w_j) = \frac{\exp(u_{w_i}^T v_{w_j})}{\sum_{l=1}^{V} u_l^T v_{w_j}}$$

其中 `V` 表示词汇数量。在 `skip-gram` 模型中使用 `softmax` 是非常昂贵的，因为计算 `log p(w_i|w_j)` 与 `V` 是成比例的。为了加快 `Word2Vec` 的训练速度， `MLlib` 使用了分层 `softmax` ,这样可以将计算的复杂度降低为 `O(log(V))` 。

2 实例

下面的例子展示了怎样加载文本数据、切分数据、构造 `Word2Vec` 实例、训练模型。最后，我们打印某个词的40个同义词。

```
import org.apache.spark._
import org.apache.spark.rdd._
import org.apache.spark.SparkContext._
import org.apache.spark.mllib.feature.{Word2Vec, Word2VecModel}
val input = sc.textFile("text8").map(line => line.split(" ").toSeq)
val word2vec = new Word2Vec()
val model = word2vec.fit(input)
val synonyms = model.findSynonyms("china", 40)
for((synonym, cosineSimilarity) <- synonyms) {
  println(s"$synonym $cosineSimilarity")
}
```

3 源码分析

由于涉及神经网络相关的知识，这里先不作分析，后续会补上。要更详细了解 `Word2Vector` 可以阅读文献【2】。

参考文献

【1】哈夫曼树与哈夫曼编码

【2】Deep Learning 实战之 word2vec

【3】Word2Vector谷歌实现

CountVectorizer

`CountVectorizer` 和 `CountVectorizerModel` 的目的是帮助我们将文本文档集转换为词频(`token counts`)向量。 当事先没有可用的词典时, `CountVectorizer` 可以被当做一个 `Estimator` 去抽取词汇,并且生成 `CountVectorizerModel` 。 这个模型通过词汇集为文档生成一个稀疏的表示,这个表示可以作为其它算法的输入,比如 `LDA` 。 在训练的过程中, `CountVectorizer` 将会选择使用语料中词频个数前 `vocabSize` 的词。一个可选的参数 `minDF` 也 会影响训练过程。这个参数表示可以包含在词典中的词的最小个数(如果该参数小于1,则表示比例)。另外一个可选的 `boolean` 参数控制着输出向量。 如果将它设置为 `true` ,那么所有的非0词频都会赋值为1。这对离散的概率模型非常有用。

举例

假设我们有下面的 `DataFrame` ,它的列名分别是 `id` 和 `texts` .

```
id  | texts
----|-------------------------------
 0  | Array("a", "b", "c")
 1  | Array("a", "b", "b", "c", "a")
```

`texts` 列的每一行表示一个类型为 `Array[String]` 的文档。 `CountVectorizer` 生成了一个带有词典 `(a, b, c)` 的 `CountVectorizerModel` 。 经过转换之后,输出的列为 `vector` 。

```
 id | texts                           | vector
----|---------------------------------|---------------
 0  | Array("a", "b", "c")            | (3,[0,1,2],[1.0,1.0,1.0]
)
 1  | Array("a", "b", "b", "c", "a")  | (3,[0,1,2],[2.0,2.0,1.0]
)
```

下面是代码调用的方法。

```
import org.apache.spark.ml.feature.{CountVectorizer, CountVectorizerModel}

val df = spark.createDataFrame(Seq(
  (0, Array("a", "b", "c")),
  (1, Array("a", "b", "b", "c", "a"))
)).toDF("id", "words")

// fit a CountVectorizerModel from the corpus
val cvModel: CountVectorizerModel = new CountVectorizer()
  .setInputCol("words")
  .setOutputCol("features")
  .setVocabSize(3)
  .setMinDF(2)
  .fit(df)

// alternatively, define CountVectorizerModel with a-priori vocabulary
val cvm = new CountVectorizerModel(Array("a", "b", "c"))
  .setInputCol("words")
  .setOutputCol("features")

cvModel.transform(df).select("features").show()
```

规则化

规则化器缩放单个样本让其拥有单位L^{p}范数。这是文本分类和聚类常用的操作。例如，两个L^{2}规则化的 `TFIDF` 向量的点乘就是两个向量的 `cosine` 相似度。

`Normalizer` 实现 `VectorTransformer` ，将一个向量规则化为转换的向量，或者将一个 `RDD` 规则化为另一个 `RDD` 。下面是一个规则化的例子。

```
import org.apache.spark.SparkContext._
import org.apache.spark.mllib.feature.Normalizer
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.util.MLUtils
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_
data.txt")
//默认情况下，p=2。计算2阶范数
val normalizer1 = new Normalizer()
val normalizer2 = new Normalizer(p = Double.PositiveInfinity)
// Each sample in data1 will be normalized using $L^2$ norm.
val data1 = data.map(x => (x.label, normalizer1.transform(x.feat
ures)))
// Each sample in data2 will be normalized using $L^\infty$ norm.

val data2 = data.map(x => (x.label, normalizer2.transform(x.feat
ures)))
```

规则化的实现很简单，我们看它的 `transform` 方法。

```
 override def transform(vector: Vector): Vector = {
    //求范数
    val norm = Vectors.norm(vector, p)
    if (norm != 0.0) {
      //稀疏向量可以重用index
      vector match {
        case DenseVector(vs) =>
          val values = vs.clone()
          val size = values.size
          var i = 0
          while (i < size) {
            values(i) /= norm
            i += 1
          }
          Vectors.dense(values)
        case SparseVector(size, ids, vs) =>
          val values = vs.clone()
          val nnz = values.size
          var i = 0
          while (i < nnz) {
            values(i) /= norm
            i += 1
          }
          Vectors.sparse(size, ids, values)
        case v => throw new IllegalArgumentException("Do not sup
port vector type " + v.getClass)
      }
    } else {
      vector
    }
  }
```

求范数调用了 `Vectors.norm` 方法，我们可以看看该方法的实现。

```
def norm(vector: Vector, p: Double): Double = {
    val values = vector match {
      case DenseVector(vs) => vs
      case SparseVector(n, ids, vs) => vs
      case v => throw new IllegalArgumentException("Do not suppo
```

```
rt vector type " + v.getClass)
    }
    val size = values.length
    if (p == 1) {
      var sum = 0.0
      var i = 0
      while (i < size) {
        sum += math.abs(values(i))
        i += 1
      }
      sum
    } else if (p == 2) {
      var sum = 0.0
      var i = 0
      while (i < size) {
        sum += values(i) * values(i)
        i += 1
      }
      math.sqrt(sum)
    } else if (p == Double.PositiveInfinity) {
      var max = 0.0
      var i = 0
      while (i < size) {
        val value = math.abs(values(i))
        if (value > max) max = value
        i += 1
      }
      max
    } else {
      var sum = 0.0
      var i = 0
      while (i < size) {
        sum += math.pow(math.abs(values(i)), p)
        i += 1
      }
      math.pow(sum, 1.0 / p)
    }
  }
```

这里分四种情况。当 `p=1` 时，即计算一阶范数，它的值为所有元素绝对值之和。当 `p=2` 时，它的值为所有元素的平方和。当 `p == Double.PositiveInfinity` 时，返回所有元素绝对值的最大值。 如果以上三种情况都不满足，那么按照下面的公式计算。

$$(\sum_{i=1}^{n}|value_i|^p)^{\frac{1}{p}}$$

Tokenizer

Tokenization是一个将文本(如一个句子)转换为个体单元(如词)的处理过程。 一个简单的 `Tokenizer` 类就提供了这个功能。下面的例子展示了如何将句子转换为此序列。

`RegexTokenizer` 基于正则表达式匹配提供了更高级的断词(`tokenization`)。默认情况下,参数 `pattern` (默认是 `\s+`)作为分隔符, 用来切分输入文本。用户可以设置 `gaps` 参数为 `false` 用来表明正则参数 `pattern` 表示 `tokens` 而不是 `splitting gaps` ,这个类可以找到所有匹配的事件并作为结果返回。下面是调用的例子。

```
import org.apache.spark.ml.feature.{RegexTokenizer, Tokenizer}

val sentenceDataFrame = spark.createDataFrame(Seq(
  (0, "Hi I heard about Spark"),
  (1, "I wish Java could use case classes"),
  (2, "Logistic,regression,models,are,neat")
)).toDF("label", "sentence")

val tokenizer = new Tokenizer().setInputCol("sentence").setOutpu
tCol("words")
val regexTokenizer = new RegexTokenizer()
  .setInputCol("sentence")
  .setOutputCol("words")
  .setPattern("\\W") // alternatively .setPattern("\\w+").setGap
s(false)

val tokenized = tokenizer.transform(sentenceDataFrame)
tokenized.select("words", "label").take(3).foreach(println)
val regexTokenized = regexTokenizer.transform(sentenceDataFrame)
regexTokenized.select("words", "label").take(3).foreach(println)
```

StopWordsRemover

Stop words是那些需要从输入数据中排除掉的词。删除这些词的原因是，这些词出现频繁,并没有携带太多有意义的信息。

`StopWordsRemover` 输入一串句子,将这些输入句子中的停用词全部删掉。停用词列表是通过 `stopWords` 参数来指定的。 一些语言的默认停用词可以通过调用 `StopWordsRemover.loadDefaultStopWords(language)` 来获得。可以用的语言选项有 `danish` , `dutch` , `english` , `finnish` , `french` , `german` , `hungarian` , `italian` , `norwegian` , `portuguese` , `russian` , `spanish` , `swedish` 以及 `turkish` 。参数 `caseSensitive` 表示是否对大小写敏感,默认为 `false` 。

例子

假设我们有下面的 `DataFrame` ,列名为 `id` 和 `raw` 。

```
 id | raw
----|----------
 0  | [I, saw, the, red, baloon]
 1  | [Mary, had, a, little, lamb]
```

把 `raw` 作为输入列, `filtered` 作为输出列,通过应用 `StopWordsRemover` 我们可以得到下面的结果。

```
 id | raw                         | filtered
----|-----------------------------|--------------------
 0  | [I, saw, the, red, baloon]  |  [saw, red, baloon]
 1  | [Mary, had, a, little, lamb]|[Mary, little, lamb]
```

下面是代码调用的例子。

```
import org.apache.spark.ml.feature.StopWordsRemover

val remover = new StopWordsRemover()
  .setInputCol("raw")
  .setOutputCol("filtered")

val dataSet = spark.createDataFrame(Seq(
  (0, Seq("I", "saw", "the", "red", "baloon")),
  (1, Seq("Mary", "had", "a", "little", "lamb"))
)).toDF("id", "raw")

remover.transform(dataSet).show()
```

n-gram

一个n-gram是一个包含 `n` 个 `tokens` (如词)的序列。 `NGram` 可以将输入特征 转换为 `n-grams` 。

`NGram` 输入一系列的序列,参数 `n` 用来决定每个 `n-gram` 的词个数。输出包含一个 `n-grams` 序列,每个 `n-gram` 表示一个划定空间的连续词序列。 如果输入序列包含的词少于 `n` ,将不会有输出。

```
import org.apache.spark.ml.feature.NGram

val wordDataFrame = spark.createDataFrame(Seq(
  (0, Array("Hi", "I", "heard", "about", "Spark")),
  (1, Array("I", "wish", "Java", "could", "use", "case", "classe
s")),
  (2, Array("Logistic", "regression", "models", "are", "neat"))
)).toDF("label", "words")

val ngram = new NGram().setInputCol("words").setOutputCol("ngram
s")
val ngramDataFrame = ngram.transform(wordDataFrame)
ngramDataFrame.take(3).map(_.getAs[Stream[String]]("ngrams").toL
ist).foreach(println)
```

Binarizer

`Binarization` 是一个将数值特征转换为二值特征的处理过程。 `threshold` 参数表示决定二值化的阈值。 值大于阈值的特征二值化为1,否则二值化为0。下面是代码调用的例子。

```
import org.apache.spark.ml.feature.Binarizer

val data = Array((0, 0.1), (1, 0.8), (2, 0.2))
val dataFrame = spark.createDataFrame(data).toDF("label", "featu
re")

val binarizer: Binarizer = new Binarizer()
  .setInputCol("feature")
  .setOutputCol("binarized_feature")
  .setThreshold(0.5)

val binarizedDataFrame = binarizer.transform(dataFrame)
val binarizedFeatures = binarizedDataFrame.select("binarized_fea
ture")
binarizedFeatures.collect().foreach(println)
```

PolynomialExpansion(多元展开)

Polynomial expansion是一个将特征展开到多元空间的处理过程。 它通过 `n-degree` 结合原始的维度来定义。比如设置 `degree` 为2就可以将 `(x, y)` 转化为 `(x, x x, y, x y, y y)` 。 `PolynomialExpansion` 提供了这个功能。 下面的例子展示了如何将特征展开为一个 `3-degree` 多项式空间。

```
import org.apache.spark.ml.feature.PolynomialExpansion
import org.apache.spark.ml.linalg.Vectors

val data = Array(
  Vectors.dense(-2.0, 2.3),
  Vectors.dense(0.0, 0.0),
  Vectors.dense(0.6, -1.1)
)
val df = spark.createDataFrame(data.map(Tuple1.apply)).toDF("fea
tures")
val polynomialExpansion = new PolynomialExpansion()
  .setInputCol("features")
  .setOutputCol("polyFeatures")
  .setDegree(3)
val polyDF = polynomialExpansion.transform(df)
polyDF.select("polyFeatures").take(3).foreach(println)
```

Discrete Cosine Transform (DCT)

Discrete Cosine Transform将一个在时间域(`time domain`)内长度为 `N` 的实值序列转换为另外一个 在频率域(`frequency domain`)内的长度为 `N` 的实值序列。下面是程序调用的例子。

```
import org.apache.spark.ml.feature.DCT
import org.apache.spark.ml.linalg.Vectors

val data = Seq(
  Vectors.dense(0.0, 1.0, -2.0, 3.0),
  Vectors.dense(-1.0, 2.0, 4.0, -7.0),
  Vectors.dense(14.0, -2.0, -5.0, 1.0))

val df = spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")

val dct = new DCT()
  .setInputCol("features")
  .setOutputCol("featuresDCT")
  .setInverse(false)

val dctDf = dct.transform(df)
dctDf.select("featuresDCT").show(3)
```

StringIndexer

`StringIndexer` 将标签列的字符串编码为标签索引。这些索引是 `[0,numLabels)` ,通过标签频率排序,所以频率最高的标签的索引为0。 如果输入列是数字,我们把它强转为字符串然后在编码。

例子

假设我们有下面的 `DataFrame` ,它的列名是 `id` 和 `category` 。

```
 id | category
----|----------
 0  | a
 1  | b
 2  | c
 3  | a
 4  | a
 5  | c
```

`category` 是字符串列,拥有三个标签 `a,b,c` 。把 `category` 作为输入列, `categoryIndex` 作为输出列,使用 `StringIndexer` 我们可以得到下面的结果。

```
 id | category | categoryIndex
----|----------|---------------
 0  | a        | 0.0
 1  | b        | 2.0
 2  | c        | 1.0
 3  | a        | 0.0
 4  | a        | 0.0
 5  | c        | 1.0
```

`a` 的索引号为0是因为它的频率最高,c次之,b最后。

另外, `StringIndexer` 处理未出现的标签的策略有两个:

- 抛出一个异常(默认情况)
- 跳过出现该标签的行

让我们回到上面的例子,但是这次我们重用上面的 `StringIndexer` 到下面的数据集。

```
 id | category
----|----------
 0  | a
 1  | b
 2  | c
 3  | d
```

如果我们没有为 `StringIndexer` 设置怎么处理未见过的标签或者设置为 `error` ,它将抛出异常,否则若设置为 `skip` ,它将得到下面的结果。

```
id  | category | categoryIndex
----|----------|---------------
 0  | a        | 0.0
 1  | b        | 2.0
 2  | c        | 1.0
```

下面是程序调用的例子。

```
import org.apache.spark.ml.feature.StringIndexer

val df = spark.createDataFrame(
  Seq((0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c"))
)
).toDF("id", "category")

val indexer = new StringIndexer()
  .setInputCol("category")
  .setOutputCol("categoryIndex")

val indexed = indexer.fit(df).transform(df)
indexed.show()
```

IndexToString

与 `StringIndexer` 相对的是, `IndexToString` 将标签索引列映射回原来的字符串标签。一个通用的使用案例是使用 `StringIndexer` 将标签转换为索引,然后通过索引训练模型,最后通过 `IndexToString` 将预测的标签索引恢复成字符串标签。

例子

假设我们有下面的 `DataFrame` ,它的列名为 `id` 和 `categoryIndex` 。

```
 id | categoryIndex
----|---------------
 0  | 0.0
 1  | 2.0
 2  | 1.0
 3  | 0.0
 4  | 0.0
 5  | 1.0
```

把 `categoryIndex` 作为输入列, `originalCategory` 作为输出列,使用 `IndexToString` 我们可以恢复原来的标签。

```
id  | categoryIndex | originalCategory
----|---------------|-----------------
 0  | 0.0           | a
 1  | 2.0           | b
 2  | 1.0           | c
 3  | 0.0           | a
 4  | 0.0           | a
 5  | 1.0           | c
```

下面是程序调用的例子。

```
import org.apache.spark.ml.feature.{IndexToString, StringIndexer
}

val df = spark.createDataFrame(Seq(
  (0, "a"),
  (1, "b"),
  (2, "c"),
  (3, "a"),
  (4, "a"),
  (5, "c")
)).toDF("id", "category")

val indexer = new StringIndexer()
  .setInputCol("category")
  .setOutputCol("categoryIndex")
  .fit(df)
val indexed = indexer.transform(df)

val converter = new IndexToString()
  .setInputCol("categoryIndex")
  .setOutputCol("originalCategory")

val converted = converter.transform(indexed)
converted.select("id", "originalCategory").show()
```

OneHotEncoder

One-hot encoding将标签索引列映射为二值向量,这个向量至多有一个1值。 这个编码允许要求连续特征的算法(如逻辑回归)使用类别特征。下面是程序调用的例子。

```
import org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer
}

val df = spark.createDataFrame(Seq(
  (0, "a"),
  (1, "b"),
  (2, "c"),
  (3, "a"),
  (4, "a"),
  (5, "c")
)).toDF("id", "category")

val indexer = new StringIndexer()
  .setInputCol("category")
  .setOutputCol("categoryIndex")
  .fit(df)
val indexed = indexer.transform(df)

val encoder = new OneHotEncoder()
  .setInputCol("categoryIndex")
  .setOutputCol("categoryVec")
val encoded = encoder.transform(indexed)
encoded.select("id", "categoryVec").show()
```

VectorIndexer

`VectorIndexer` 把数据集中的类型特征索引为向量。它不仅可以自动的判断哪些特征是可以类别化,也能将原有的值转换为类别索引。 通常情况下,它的过程如下:

- 1 拿到类型为 `vector` 的输入列和参数 `maxCategories`
- 2 根据有区别的值的数量,判断哪些特征可以类别化。拥有的不同值的数量至少要为 `maxCategories` 的特征才能判断可以类别化。
- 3 对每一个可以类别化的特征计算基于0的类别索引。
- 4 为类别特征建立索引,将原有的特征值转换为索引。

索引类别特征允许诸如决策树和集合树等算法适当处理可分类化的特征,提高效率。

在下面的例子中,我们从数据集中读取标签点,然后利用 `VectorIndexer` 去判断哪些特征可以被认为是可分类化的。 我们将可分类特征的值转换为索引。转换后的数据可以传递给 `DecisionTreeRegressor` 等可以操作分类特征的算法。

```
import org.apache.spark.ml.feature.VectorIndexer

val data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

val indexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexed")
  .setMaxCategories(10)

val indexerModel = indexer.fit(data)

val categoricalFeatures: Set[Int] = indexerModel.categoryMaps.keys.toSet
println(s"Chose ${categoricalFeatures.size} categorical features: " +
  categoricalFeatures.mkString(", "))

// Create new column "indexed" with categorical values transformed to indices
val indexedData = indexerModel.transform(data)
indexedData.show()
```

特征缩放

特征缩放是用来统一资料中的自变项或特征范围的方法，在资料处理中，通常会被使用在资料前处理这个步骤。

1 动机

因为在原始的资料中，各变数的范围大不相同。对于某些机器学习的算法，若没有做过标准化，目标函数会无法适当的运作。举例来说，多数的分类器利用两点间的距离计算两点的差异， 若其中一个特征具有非常广的范围，那两点间的差异就会被该特征左右，因此，所有的特征都该被标准化，这样才能大略的使各特征依比例影响距离。另外一个做特征缩放的理由是他能使加速梯度下降法的收敛。

2 方法

2.1 重新缩放

最简单的方式是重新缩放特征的范围到 `[0, 1]` 或 `[-1, 1]` ， 依据原始的资料选择目标范围，通式如下：

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

2.2 标准化

在机器学习中，我们可能要处理不同种类的资料，例如，音讯和图片上的像素值，这些资料可能是高维度的，资料标准化后会使每个特征中的数值平均变为0(将每个特征的值都减掉原始资料中该特征的平均)、标准差变为1，这个方法被广泛的使用在许多机器学习算法中。

3 实例

```
import org.apache.spark.SparkContext._
import org.apache.spark.mllib.feature.StandardScaler
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.util.MLUtils
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_
data.txt")

val scaler1 = new StandardScaler().fit(data.map(x => x.features)
)
val scaler2 = new StandardScaler(withMean = true, withStd = true
).fit(data.map(x => x.features))
// scaler3 is an identical model to scaler2, and will produce id
entical transformations
val scaler3 = new StandardScalerModel(scaler2.std, scaler2.mean)
// data1 will be unit variance.
val data1 = data.map(x => (x.label, scaler1.transform(x.features
)))
// Without converting the features into dense vectors, transform
ation with zero mean will raise
// exception on sparse vector.
// data2 will be unit variance and zero mean.
val data2 = data.map(x => (x.label, scaler2.transform(Vectors.de
nse(x.features.toArray))))
```

4 源代码实现

在 `MLlib` 中， `StandardScaler` 类用于标准化特征。

```
class StandardScaler @Since("1.1.0") (withMean: Boolean, withStd
: Boolean)
```

`StandardScaler` 的实现中提供了两个参数 `withMean` 和 `withStd` 。在介绍这两个参数之前，我们先了解 `fit` 方法的实现。

```
def fit(data: RDD[Vector]): StandardScalerModel = {
    // TODO: skip computation if both withMean and withStd are f
alse
    val summary = data.treeAggregate(new MultivariateOnlineSumma
rizer)(
      (aggregator, data) => aggregator.add(data),
      (aggregator1, aggregator2) => aggregator1.merge(aggregator
2))
    new StandardScalerModel(
      Vectors.dense(summary.variance.toArray.map(v => math.sqrt(
v))),
      summary.mean,
      withStd,
      withMean)
  }
```

该方法计算数据集的均值和方差（查看概括统计以了解更多信息），并初始化 `StandardScalerModel` 。初始化 `StandardScalerModel` 之后，我们就可以调用 `transform` 方法转换特征了。

当 `withMean` 参数为 `true` 时， `transform` 的实现如下。

```
private lazy val shift: Array[Double] = mean.toArray
val localShift = shift
      vector match {
        case DenseVector(vs) =>
          val values = vs.clone()
          val size = values.size
          if (withStd) {
            var i = 0
            while (i < size) {
              values(i) = if (std(i) != 0.0) (values(i) - localS
hift(i)) * (1.0 / std(i)) else 0.0
              i += 1
            }
          } else {
            var i = 0
            while (i < size) {
              values(i) -= localShift(i)
              i += 1
            }
          }
          Vectors.dense(values)
        case v => throw new IllegalArgumentException("Do not sup
port vector type " + v.getClass)
}
```

以上代码显示，当 `withMean` 为 `true` ， `withStd` 为 `false` 时，向量中的各元素均减去它相应的均值。当 `withMean` 和 `withStd` 均为 `true` 时，各元素在减去相应的均值之后，还要除以它们相应的方差。 当 `withMean` 为 `true` ，程序只能处理稠密的向量，不能处理稀疏向量。

当 `withMean` 为 `false` 时， `transform` 的实现如下。

```
 vector match {
        case DenseVector(vs) =>
          val values = vs.clone()
          val size = values.size
          var i = 0
          while(i < size) {
            values(i) *= (if (std(i) != 0.0) 1.0 / std(i) else 0
.0)
            i += 1
          }
          Vectors.dense(values)
        case SparseVector(size, indices, vs) =>
          // For sparse vector, the `index` array inside sparse
vector object will not be changed,
          // so we can re-use it to save memory.
          val values = vs.clone()
          val nnz = values.size
          var i = 0
          while (i < nnz) {
            values(i) *= (if (std(indices(i)) != 0.0) 1.0 / std(
indices(i)) else 0.0)
            i += 1
          }
          Vectors.sparse(size, indices, values)
        case v => throw new IllegalArgumentException("Do not sup
port vector type " + v.getClass)
 }
```

这里的处理很简单，就是将数据集的列的标准差归一化为1。

参考文献

【1】特征缩放

MinMaxScaler

`MinMaxScaler` 转换由向量行组成的数据集,将每个特征调整到一个特定的范围(通常是 `[0,1]`)。它有下面两个参数:

- `min` :默认是0。转换的下界,被所有的特征共享。
- `max` :默认是1。转换的上界,被所有特征共享。

`MinMaxScaler` 计算数据集上的概要统计数据,产生一个 `MinMaxScalerModel` 。然后就可以用这个模型单独的转换每个特征到特定的范围。 特征 `E` 被转换后的值可以用下面的公式计算:

$$\frac{e_{i} - E_{min}}{E_{max} - E_{min}} * (max - min) + min$$

对于 `E_{max} == E_{min}` 的情况, `Rescaled(e_i) = 0.5 * (max + min)` 。

注意,由于0值有可能转换成非0的值,所以转换的输出为 `DenseVector` ,即使输入为稀疏的数据也一样。下面的例子展示了如何将特征转换到 `[0,1]` 。

```
import org.apache.spark.ml.feature.MinMaxScaler

val dataFrame = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

val scaler = new MinMaxScaler()
  .setInputCol("features")
  .setOutputCol("scaledFeatures")

// Compute summary statistics and generate MinMaxScalerModel
val scalerModel = scaler.fit(dataFrame)

// rescale each feature to range [min, max].
val scaledData = scalerModel.transform(dataFrame)
scaledData.show()
```

MaxAbsScaler

`MaxAbsScaler` 转换由向量列组成的数据集,将每个特征调整到 `[-1,1]` 的范围,它通过每个特征内的最大绝对值来划分。 它不会移动和聚集数据,因此不会破坏任何的稀疏性。

`MaxAbsScaler` 计算数据集上的统计数据,生成 `MaxAbsScalerModel` ,然后使用生成的模型分别的转换特征到范围 `[-1,1]` 。下面是程序调用的例子。

```
import org.apache.spark.ml.feature.MaxAbsScaler

val dataFrame = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")
val scaler = new MaxAbsScaler()
  .setInputCol("features")
  .setOutputCol("scaledFeatures")

// Compute summary statistics and generate MaxAbsScalerModel
val scalerModel = scaler.fit(dataFrame)

// rescale each feature to range [-1, 1]
val scaledData = scalerModel.transform(dataFrame)
scaledData.show()
```

Bucketizer

`Bucketizer` 将连续的特征列转换成特征桶(`buckets`)列。这些桶由用户指定。它拥有一个 `splits` 参数。

- `splits` :如果有 `n+1` 个 `splits` ,那么将有 `n` 个桶。桶将由 `split x` 和 `split y` 共同确定,它的值范围为 `[x,y)` ,如果是最后 一个桶,范围将是 `[x,y]` 。 `splits` 应该严格递增。负无穷和正无穷必须明确的提供用来覆盖所有的双精度值,否则,超出 `splits` 的值将会被 认为是一个错误。 `splits` 的两个例子是 `Array(Double.NegativeInfinity, 0.0, 1.0, Double.PositiveInfinity)` 和 `Array(0.0, 1.0, 2.0)` 。

注意,如果你并不知道目标列的上界和下界,你应该添加 `Double.NegativeInfinity` 和 `Double.PositiveInfinity` 作为边界从而防止潜在的 超过边界的异常。下面是程序调用的例子。

```
import org.apache.spark.ml.feature.Bucketizer

val splits = Array(Double.NegativeInfinity, -0.5, 0.0, 0.5, Doub
le.PositiveInfinity)

val data = Array(-0.5, -0.3, 0.0, 0.2)
val dataFrame = spark.createDataFrame(data.map(Tuple1.apply)).to
DF("features")

val bucketizer = new Bucketizer()
  .setInputCol("features")
  .setOutputCol("bucketedFeatures")
  .setSplits(splits)

// Transform original data into its bucket index.
val bucketedData = bucketizer.transform(dataFrame)
bucketedData.show()
```

元素智能乘积

`ElementwiseProduct` 对每一个输入向量乘以一个给定的“权重”向量。换句话说，就是通过一个乘子对数据集的每一列进行缩放。这个转换可以表示为如下的形式：

$$\begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} v_1 w_1 \\ \vdots \\ v_n w_n \end{pmatrix}$$

下面是一个使用的实例。

```
import org.apache.spark.SparkContext._
import org.apache.spark.mllib.feature.ElementwiseProduct
import org.apache.spark.mllib.linalg.Vectors
// Create some vector data; also works for sparse vectors
val data = sc.parallelize(Array(Vectors.dense(1.0, 2.0, 3.0), Ve
ctors.dense(4.0, 5.0, 6.0)))
val transformingVector = Vectors.dense(0.0, 1.0, 2.0)
val transformer = new ElementwiseProduct(transformingVector)

// Batch transform and per-row transform give the same results:
val transformedData = transformer.transform(data)
val transformedData2 = data.map(x => transformer.transform(x))
```

下面看 `transform` 的实现。

```
override def transform(vector: Vector): Vector = {
    vector match {
      case dv: DenseVector =>
        val values: Array[Double] = dv.values.clone()
        val dim = scalingVec.size
        var i = 0
        while (i < dim) {
          //相对应的值相乘
          values(i) *= scalingVec(i)
          i += 1
        }
        Vectors.dense(values)
      case SparseVector(size, indices, vs) =>
        val values = vs.clone()
        val dim = values.length
        var i = 0
        while (i < dim) {
          //相对应的值相乘
          values(i) *= scalingVec(indices(i))
          i += 1
        }
        Vectors.sparse(size, indices, values)
      case v => throw new IllegalArgumentException("Does not sup
port vector type " + v.getClass)
    }
  }
```

SQLTransformer

`SQLTransformer` 实现了一种转换,这个转换通过 `SQl` 语句来定义。目前我们仅仅支持的 `SQL` 语法是像 `SELECT ... FROM __THIS__ ...` 的形式。 这里 `__THIS__` 表示输入数据集相关的表。例如, `SQLTransformer` 支持的语句如下:

- `SELECT a, a + b AS a_b FROM __THIS__`
- `SELECT a, SQRT(b) AS b_sqrt FROM __THIS__ where a > 5`
- `SELECT a, b, SUM(c) AS c_sum FROM __THIS__ GROUP BY a, b`

例子

假设我们拥有下面的 `DataFrame` ,它的列名是 `id,v1,v2` 。

```
id  |  v1 |  v2
----|-----|-----
 0  | 1.0 | 3.0
 2  | 2.0 | 5.0
```

下面是语句 `SELECT *, (v1 + v2) AS v3, (v1 * v2) AS v4 FROM __THIS__` 的输出结果。

```
 id |  v1 |  v2 |  v3 |  v4
----|-----|-----|-----|-----
 0  | 1.0 | 3.0 | 4.0 | 3.0
 2  | 2.0 | 5.0 | 7.0 |10.0
```

下面是程序调用的例子。

```
import org.apache.spark.ml.feature.SQLTransformer

val df = spark.createDataFrame(
  Seq((0, 1.0, 3.0), (2, 2.0, 5.0))).toDF("id", "v1", "v2")

val sqlTrans = new SQLTransformer().setStatement(
  "SELECT *, (v1 + v2) AS v3, (v1 * v2) AS v4 FROM __THIS__")

sqlTrans.transform(df).show()
```

VectorAssembler

`VectorAssembler`是一个转换器,它可以将给定的多列转换为一个向量列。合并原始特征与通过不同的转换器转换而来的特征,从而训练机器学习模型, `VectorAssembler`是非常有用的。`VectorAssembler`允许这些类型:所有的数值类型, `boolean`类型以及`vector`类型。

例子

假设我们有下面的`DataFrame`,它的列名分别是`id, hour, mobile, userFeatures, clicked`。

```
id  | hour | mobile | userFeatures     | clicked
----|------|--------|------------------|---------
 0  | 18   | 1.0    | [0.0, 10.0, 0.5] | 1.0
```

`userFeatures`是一个向量列,包含三个用户特征。我们想合并`hour`, `mobile`和`userFeatures`到一个名为`features`的特征列。通过转换之后,我们可以得到下面的结果。

```
id  | hour | mobile | userFeatures     | clicked | features
----|------|--------|------------------|---------|--------------
---------------
 0  | 18   | 1.0    | [0.0, 10.0, 0.5] | 1.0     | [18.0, 1.0, 0
.0, 10.0, 0.5]
```

下面是程序调用的例子。

```
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors

val dataset = spark.createDataFrame(
  Seq((0, 18, 1.0, Vectors.dense(0.0, 10.0, 0.5), 1.0))
).toDF("id", "hour", "mobile", "userFeatures", "clicked")

val assembler = new VectorAssembler()
  .setInputCols(Array("hour", "mobile", "userFeatures"))
  .setOutputCol("features")

val output = assembler.transform(dataset)
println(output.select("features", "clicked").first())
```

QuantileDiscretizer

`QuantileDiscretizer` 输入连续的特征列,输出分箱的类别特征。分箱数是通过参数 `numBuckets` 来指定的。 箱的范围是通过使用近似算法(见 approxQuantile)来得到的。 近似的精度可以通过 `relativeError` 参数来控制。当这个参数设置为0时,将会计算精确的分位数。箱的上边界和下边界分别是正无穷和负无穷时, 取值将会覆盖所有的实数值。

例子

假设我们有下面的 `DataFrame` ,它的列名是 `id,hour` 。

```
 id | hour
----|------
 0  | 18.0
----|------
 1  | 19.0
----|------
 2  | 8.0
----|------
 3  | 5.0
----|------
 4  | 2.2
```

`hour` 是类型为 `DoubleType` 的连续特征。我们想将连续特征转换为一个分类特征。给定 `numBuckets` 为3,我们可以得到下面的结果。

```
id  | hour | result
----|------|------
 0  | 18.0 | 2.0
----|------|------
 1  | 19.0 | 2.0
----|------|------
 2  | 8.0  | 1.0
----|------|------
 3  | 5.0  | 1.0
----|------|------
 4  | 2.2  | 0.0
```

下面是代码实现的例子。

```
import org.apache.spark.ml.feature.QuantileDiscretizer

val data = Array((0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, 2
.2))
var df = spark.createDataFrame(data).toDF("id", "hour")

val discretizer = new QuantileDiscretizer()
  .setInputCol("hour")
  .setOutputCol("result")
  .setNumBuckets(3)

val result = discretizer.fit(df).transform(df)
result.show()
```

VectorSlicer

`VectorSlicer` 是一个转换器,输入一个特征向量输出一个特征向量,它是原特征的一个子集。这在从向量列中抽取特征非常有用。 `VectorSlicer` 接收一个拥有特定索引的特征列,它的输出是一个新的特征列,它的值通过输入的索引来选择。有两种类型的索引:

- 1、整数索引表示进入向量的索引,调用 `setIndices()`
- 2、字符串索引表示进入向量的特征列的名称,调用 `setNames()` 。这种情况需要向量列拥有一个 `AttributeGroup` ,这是因为实现是通过属性的名字来匹配的。

整数和字符串都是可以使用的,并且,整数和字符串可以同时使用。至少需要选择一个特征,而且重复的特征是不被允许的。

输出向量首先会按照选择的索引进行排序,然后再按照选择的特征名进行排序。

例子

假设我们有下面的 `DataFrame` ,它的列名是 `userFeatures` 。

```
 userFeatures
------------------
 [0.0, 10.0, 0.5]
```

`userFeatures` 是一个向量列,它包含三个用户特征。假设用户特征的第一列均为0,所以我们想删除它,仅仅选择后面的两列。 `VectorSlicer` 通过 `setIndices(1,2)` 选择后面的两项,产生下面新的名为 `features` 的向量列。

```
 userFeatures     | features
------------------|-----------------------------
 [0.0, 10.0, 0.5] | [10.0, 0.5]
```

假设我们还有潜在的输入特性,如 `["f1", "f2", "f3"]` ,我们还可以通过 `setNames("f2", "f3")` 来选择。

```
 userFeatures      | features
------------------|-----------------------------
 [0.0, 10.0, 0.5] | [10.0, 0.5]
 ["f1", "f2", "f3"] | ["f2", "f3"]
```

下面是程序调用的例子。

```
import java.util.Arrays

import org.apache.spark.ml.attribute.{Attribute, AttributeGroup,
NumericAttribute}
import org.apache.spark.ml.feature.VectorSlicer
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.StructType

val data = Arrays.asList(Row(Vectors.dense(-2.0, 2.3, 0.0)))

val defaultAttr = NumericAttribute.defaultAttr
val attrs = Array("f1", "f2", "f3").map(defaultAttr.withName)
val attrGroup = new AttributeGroup("userFeatures", attrs.asInsta
nceOf[Array[Attribute]])

val dataset = spark.createDataFrame(data, StructType(Array(attrG
roup.toStructField())))

val slicer = new VectorSlicer().setInputCol("userFeatures").setO
utputCol("features")

slicer.setIndices(Array(1)).setNames(Array("f3"))
// or slicer.setIndices(Array(1, 2)), or slicer.setNames(Array("
f2", "f3"))

val output = slicer.transform(dataset)
println(output.select("userFeatures", "features").first())
```

RFormula

`RFormula` 通过一个R model formula选择一个特定的列。 目前我们支持 `R` 算子的一个受限的子集,包括 `~` , `.` , `:` , `+` , `-` 。这些基本的算子是:

- `~` 分开 `target` 和 `terms`
- `+` 连接 `term` , `+ 0` 表示删除截距(`intercept`)
- `-` 删除 `term` , `- 1` 表示删除截距
- `:` 交集
- `.` 除了 `target` 之外的所有列

假设 `a` 和 `b` 是 `double` 列,我们用下面简单的例子来证明 `RFormula` 的有效性。

- `y ~ a + b` 表示模型 `y ~ w0 + w1 * a + w2 * b` ,其中 `w0` 是截距, `w1` 和 `w2` 是系数
- `y ~ a + b + a:b - 1` 表示模型 `y ~ w1 * a + w2 * b + w3 * a * b` ,其中 `w1` , `w2` , `w3` 是系数

`RFormula` 产生一个特征向量列和一个 `double` 或 `string` 类型的标签列。比如在线性回归中使用 `R` 中的公式时, 字符串输入列是 `one-hot` 编码,数值列强制转换为 `double` 类型。如果标签列是字符串类型,它将使用 `StringIndexer` 转换为 `double` 类型。如果 `DataFrame` 中不存在标签列,输出的标签列将通过公式中指定的返回变量来创建。

例子

假设我们有一个 `DataFrame` ,它的列名是 `id` , `country` , `hour` 和 `clicked` 。

```
id | country | hour | clicked
---|---------|------|---------
 7 | "US"    | 18   | 1.0
 8 | "CA"    | 12   | 0.0
 9 | "NZ"    | 15   | 0.0
```

如果我们用 `clicked ~ country + hour` (基于 `country` 和 `hour` 来预测 `clicked`)来作用于 `RFormula` ,将会得到下面的结果。

```
id | country | hour | clicked | features         | label
---|---------|------|---------|------------------|-------
 7 | "US"    | 18   | 1.0     | [0.0, 0.0, 18.0] | 1.0
 8 | "CA"    | 12   | 0.0     | [0.0, 1.0, 12.0] | 0.0
 9 | "NZ"    | 15   | 0.0     | [1.0, 0.0, 15.0] | 0.0
```

下面是代码调用的例子。

```
import org.apache.spark.ml.feature.RFormula

val dataset = spark.createDataFrame(Seq(
  (7, "US", 18, 1.0),
  (8, "CA", 12, 0.0),
  (9, "NZ", 15, 0.0)
)).toDF("id", "country", "hour", "clicked")
val formula = new RFormula()
  .setFormula("clicked ~ country + hour")
  .setFeaturesCol("features")
  .setLabelCol("label")
val output = formula.fit(dataset).transform(dataset)
output.select("features", "label").show()
```

卡方选择器

特征选择试图识别相关的特征用于模型构建。它改变特征空间的大小，它可以提高速度以及统计学习行为。`ChiSqSelector`实现卡方特征选择，它操作于带有类别特征的标注数据。`ChiSqSelector`根据独立的卡方测试对特征进行排序，然后选择排序最高的特征。下面是一个使用的例子。

```
import org.apache.spark.SparkContext._
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.mllib.feature.ChiSqSelector
// 加载数据
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_
data.txt")
// 卡方分布需要类别特征，所以对特征除一个整数。虽然特征是double类型，
//但是ChiSqSelector将每个唯一的值当做一个类别
val discretizedData = data.map { lp =>
  LabeledPoint(lp.label, Vectors.dense(lp.features.toArray.map {
 x => (x / 16).floor } ) )
}
// Create ChiSqSelector that will select top 50 of 692 features
val selector = new ChiSqSelector(50)
// Create ChiSqSelector model (selecting features)
val transformer = selector.fit(discretizedData)
// Filter the top 50 features from each feature vector
val filteredData = discretizedData.map { lp =>
  LabeledPoint(lp.label, transformer.transform(lp.features))
}
```

下面看看选择特征的实现，入口函数是`fit`。

```
def fit(data: RDD[LabeledPoint]): ChiSqSelectorModel = {
    //计算数据卡方值
    val indices = Statistics.chiSqTest(data)
      .zipWithIndex.sortBy { case (res, _) => -res.statistic }
      .take(numTopFeatures)
      .map { case (_, indices) => indices }
      .sorted
    new ChiSqSelectorModel(indices)
  }
```

这里通过 `Statistics.chiSqTest` 计算卡方检测的值。下面需要了解卡方检测的理论基础。

1 卡方检测

1.1 什么是卡方检测

卡方检验是一种用途很广的计数资料的假设检验方法。它属于非参数检验的范畴，主要是比较两个及两个以上样本率(构成比）以及两个分类变量的关联性分析。 其根本思想就是在于比较理论频数和实际频数的吻合程度或拟合优度问题。

1.2 卡方检测的基本思想

卡方检验是以${X}^{2}$分布为基础的一种常用假设检验方法，它的无效假设 `H0` 是：观察频数与期望频数没有差别。

该检验的基本思想是：首先假设 `H0` 成立，基于此前提计算出${X}^{2}$值，它表示观察值与理论值之间的偏离程度。根据${X}^{2}$分布及自由度可以确定在 `H0` 假设成立的情况下获得当前统计量及更极端情况的概率 `P` 。 如果P值很小，说明观察值与理论值偏离程度太大，应当拒绝无效假设，表示比较资料之间有显著差异；否则就不能拒绝无效假设，尚不能认为样本所代表的实际情况和理论假设有差别。

1.3 卡方值的计算与意义

卡方值表示观察值与理论值之间的偏离程度。计算这种偏离程度的基本思路如下。

- 设 `A` 代表某个类别的观察频数， `E` 代表基于 `H0` 计算出的期望频数， `A` 与 `E` 之差称为残差。
- 残差可以表示某一个类别观察值和理论值的偏离程度，但如果将残差简单相加以表示各类别观察频数与期望频数的差别，则有一定的不足之处。 因为残差有正有负，相加后会彼此抵消，总和仍然为0，为此可以将残差平方后求和。
- 另一方面，残差大小是一个相对的概念，相对于期望频数为10时，期望频数为20的残差非常大，但相对于期望频数为1000时20的残差就很小了。 考虑到这一点，人们又将残差平方除以期望频数再求和，以估计观察频数与期望频数的差别。

进行上述操作之后，就得到了常用的${X}^{2}$统计量。其计算公式是：

$$\chi_2 = \sum \frac{(A-E)^2}{E} = \sum_{i=1}^{k} \frac{(A_i - E_i)^2}{E_i} = \sum_{i=1}^{k} \frac{(A_i - np_i)^2}{np_i}$$

当 `n` 比较大时，卡方统计量近似服从 `k-1` (计算 `E_i` 时用到的参数个数)个自由度的卡方分布。由卡方的计算公式可知，当观察频数与期望频数完全一致时，卡方值为0；观察频数与期望频数越接近，两者之间的差异越小，卡方值越小； 反之，观察频数与期望频数差别越大，两者之间的差异越大，卡方值越大。

2 卡方检测的源码实现

在 `MLlib` 中，使用 `chiSquaredFeatures` 方法实现卡方检测。它为每个特征进行皮尔森独立检测。下面看它的代码实现。

```
def chiSquaredFeatures(data: RDD[LabeledPoint],
      methodName: String = PEARSON.name): Array[ChiSqTestResult]
  = {
    val maxCategories = 10000
    val numCols = data.first().features.size
    val results = new Array[ChiSqTestResult](numCols)
    var labels: Map[Double, Int] = null
    // 某个时刻至少1000列
```

```
    val batchSize = 1000
    var batch = 0
    while (batch * batchSize < numCols) {
      val startCol = batch * batchSize
      val endCol = startCol + math.min(batchSize, numCols - star
tCol)
      val pairCounts = data.mapPartitions { iter =>
        val distinctLabels = mutable.HashSet.empty[Double]
        val allDistinctFeatures: Map[Int, mutable.HashSet[Double
]] =
          Map((startCol until endCol).map(col => (col, mutable.H
ashSet.empty[Double])): _*)
        var i = 1
        iter.flatMap { case LabeledPoint(label, features) =>
          if (i % 1000 == 0) {
            if (distinctLabels.size > maxCategories) {
              throw new SparkException
            }
            allDistinctFeatures.foreach { case (col, distinctFea
tures) =>
              if (distinctFeatures.size > maxCategories) {
                throw new SparkException
              }
            }
          }
          i += 1
          distinctLabels += label
          features.toArray.view.zipWithIndex.slice(startCol, end
Col).map { case (feature, col) =>
            allDistinctFeatures(col) += feature
            (col, feature, label)
          }
        }
      }.countByValue()
      if (labels == null) {
        // Do this only once for the first column since labels a
re invariant across features.
        labels =
          pairCounts.keys.filter(_._1 == startCol).map(_._3).toA
rray.distinct.zipWithIndex.toMap
```

```
        }
        val numLabels = labels.size
        pairCounts.keys.groupBy(_._1).map { case (col, keys) =>
          val features = keys.map(_._2).toArray.distinct.zipWithIn
dex.toMap
          val numRows = features.size
          val contingency = new BDM(numRows, numLabels, new Array[
Double](numRows * numLabels))
          keys.foreach { case (_, feature, label) =>
            val i = features(feature)
            val j = labels(label)
            //带有标签的特征的出现次数
            contingency(i, j) += pairCounts((col, feature, label))
          }
          results(col) = chiSquaredMatrix(Matrices.fromBreeze(cont
ingency), methodName)
        }
        batch += 1
      }
      results
    }
```

上述代码主要对数据进行处理，获取带有标签的特征的出现次数，并用这个次数计算卡方值。真正获取卡方值的函数是 `chiSquaredMatrix` 。

```
 def chiSquaredMatrix(counts: Matrix, methodName: String = PEARS
ON.name): ChiSqTestResult = {
    val method = methodFromString(methodName)
    val numRows = counts.numRows
    val numCols = counts.numCols
    // get row and column sums
    val colSums = new Array[Double](numCols)
    val rowSums = new Array[Double](numRows)
    val colMajorArr = counts.toArray
    val colMajorArrLen = colMajorArr.length
    var i = 0
    while (i < colMajorArrLen) {
      val elem = colMajorArr(i)
      if (elem < 0.0) {
```

```
        throw new IllegalArgumentException("Contingency table ca
nnot contain negative entries.")
      }
      //每列的总数
      colSums(i / numRows) += elem
      //每行的总数
      rowSums(i % numRows) += elem
      i += 1
    }
    //所有元素的总和
    val total = colSums.sum
    // second pass to collect statistic
    var statistic = 0.0
    var j = 0
    while (j < colMajorArrLen) {
      val col = j / numRows
      val colSum = colSums(col)
      if (colSum == 0.0) {
        throw new IllegalArgumentException("Chi-squared statisti
c undefined for input matrix due to"
          + s"0 sum in column [$col].")
      }
      val row = j % numRows
      val rowSum = rowSums(row)
      if (rowSum == 0.0) {
        throw new IllegalArgumentException("Chi-squared statisti
c undefined for input matrix due to"
          + s"0 sum in row [$row].")
      }
      //期望值
      val expected = colSum * rowSum / total
      //PEARSON
      statistic += method.chiSqFunc(colMajorArr(j), expected)
      j += 1
    }
    //自由度
    val df = (numCols - 1) * (numRows - 1)
    if (df == 0) {
      // 1 column or 1 row. Constant distribution is independent
 of anything.
```

```
        // pValue = 1.0 and statistic = 0.0 in this case.
        new ChiSqTestResult(1.0, 0, 0.0, methodName, NullHypothesis
.independence.toString)
      } else {
        //计算累积概率
        val pValue = 1.0 - new ChiSquaredDistribution(df).cumulati
veProbability(statistic)
        new ChiSqTestResult(pValue, df, statistic, methodName, Nul
lHypothesis.independence.toString)
      }
    }
    //上述代码中的method.chiSqFunc(colMajorArr(j), expected)，调用下面
的代码
    val PEARSON = new Method("pearson", (observed: Double, expecte
d: Double) => {
        val dev = observed - expected
        dev * dev / expected
      })
```

上述代码的实现和参考文献【2】中 `Test of independence` 的描述一致。

参考文献

【1】卡方检验

【2】Pearson's chi-squared test